

ERDŐS IVÁN

**COMMODORE 64 ASSEMBLY**

LSI Alkalmazástechnikai  
Tanácsadó Szolgálat



A szövetkezet saját fejlesztésű termékcsaládja a FLOTISZ.

Tagjai: floppy fej-tisztító

mágnesszalag egység fej-tisztító

mágneslemezcsomag tisztító

képernyő tisztító és

írógéptisztító készlete.

Hasonló elven alapuló használatuk egyszerű, higiénikus, mint például a

floppy fej-tisztítóké,

amelyek az 5 1/4"-os és 8"-os egységek író-olvasó fejét tisztítják, ezáltal a lemezek élettartamát is megnövelik.

Egy- és kétféjes lemezegységekhez egyaránt használható.

A készlet tartalmaz 1 sprayt és 2 tisztítólemezt.

A sprayvel bepermetezzük a megadott helyen a tisztítólemezt, azt az egységbe helyezzük, és kb. 30 másodpercig olvastatjuk. Az átlagosan igénybevett gépeknél ezt elegendő havonta egyszer elvégezni.

1 db FLOTISZ készlet 2 évig használható.

**Gyártja:**



**Műszaki Fejlesztő Kiszövetkezet**  
**1136 Budapest, Fürst Sándor u. 5.**  
**Telefon: 128-128**

**ERDŐS IVÁN**

**COMMODORE 64 ASSEMBLY**  
**nyelvű programozása**

**LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT**  
**BUDAPEST, 1985**

**„A MIKROSZÁMÍTÓGÉPEK ÉS ALKALMAZÁSI RENDSZEREIK  
KUTATÁSA-FEJLESZTÉSE”**

**című**

**OMFB TÁRCAPROGRAM**

**4/b alprogramjához kapcsolódóan készült**

**Lektorálta: dr. Úry László**

**Témafelelős: SZIKLAI KLÁRA**

**ISBN 963 592 483 6**

# T A R T A L O M

OLDAL

1. Bevezetés	
1.1 Programozási nyelvek mikroszámítógépen	3
1.2 Utmutató a 2.-5. fejezethez	8
2. Mit tud a 6510-es mikroprocesszor. Programozási modell	10
2.1 Néhány alapfogalom	10
2.2 Regiszterek és jelzők	13
2.3 Címzés módok	17
2.4 Megszakítások	23
3. Assembly = kényelmes gépi kód	25
3.1 Mit tud egy assembler - mit kellene tudni egy assemblernek	25
3.2 Utasításkészlet	27
3.2.1 Töltőutasítások	28
3.2.2 Tárolóutasítások	28
3.2.3 Regiszter-utasítások	29
3.2.4 Vezérlésátadó utasítások	31
3.2.5 Veremkezelő utasítások	34
3.2.6 Léptető utasítások	34
3.2.7 Logikai utasítások	35
3.2.8 Aritmetikai utasítások	37
3.2.9 Összehasonlító utasítások	37
4. Software-eszközök - assemblerek, monitorok, debugerek	39
4.1 A HELP+ assembly programozást segítő eszközei	40
4.2 Makróassembler fejlesztő rendszer C-64-re	44
4.3 PROFIMAT	53
4.4 SM-KIT 64	58
4.5 PTD interaktív debugger	62
4.5.1 A debugger betöltése	63
4.5.2 Alapfogalmak	63
4.5.3 Program-módu végrehajtás	64
4.5.4 Szimbólumok	64
4.5.5 DOS kiegészítés	65
4.5.6 Memóriakezelés	66
4.5.7 Program-nyomkövetés	69
4.5.8 Csak program-módban használható utasítások	74
4.5.9 Kód-foltozás	77
4.5.10 A PTD debugger értékelése	79

5. Mit programozzunk, hogy programozzunk assembly nyelven?	80
5.1 Memóriagazdálkodás	81
5.2 Assembly = szerelés	93
5.3 A Commodore 64 "operációs rendszere", a KERNAL	105
5.4 BASIC és assembly békés egymás mellett élése	108
5.5 Gyorsabban, messzebbre, magasabbra...	116
5.6 Nagy feladatok megközelítése	126
Függelékek	
A. függelék C 64 kódtábla	130
B. függelék A MOS 6510 utasításkészlete	136
C. függelék KERNAL szubrutinok	139
Irodalom	146



## 1.1 Programozási nyelvek mikroszámítógépen

Az olvasó egy olyan könyvet tart a kezében, amelynek célja egy bizonyos mikroszámítógép egy bizonyos programozási nyelvének ismertetése. Mielőtt azonban szűken vett témánkra térnénk, talán nem felesleges pár szót szólni általában a programozási nyelvekről. A mikroszámítógépekről általában nem érdemes ezen a helyen értekezni, hiszen mindent és mindennek az ellenkezőjét elmondták, leírák már erről a témáról. Szógezzünk le kissé homályosan mindössze csak annyit, hogy itt és most mikroszámítógépen a Commodore 64 kategóriájú egyszerű személyi számítógépet értjük.

A programozási nyelvek története nem egyidős a számítógéppel, hanem annál messzebbre nyulik vissza. Ennek ellenére az első számítógépeket még közvetlenül gépikódban, tehát a processzor által közvetlenül megérthető kódok felhasználásával programozták. A mai értelemben vett programozási nyelvek kialakulása az ötvenes évek második felében kezdődött meg igazán, amikor az első magasszintű nyelveket, az ALGOL-t, a COBOL-t, a FORTRAN-t definiálták. Nem kívánunk azonban történelemmel foglalkozni, lássuk a mai helyzetet.

Ma a programozási nyelvek területén az elmúlt 30 évben lezajlott forradalmi fejlődés ellenére, vagy talán éppen annak következtében a helyzet elég zavaros. Igen sok programozási nyelvet használnak, és ezek néha a legkevesbé sem hasonlítanak egymásra. Mégis próbáljuk meg meghatározni, és egy aránylag elfogadott módon osztályozni őket.

Valamennyi programozási nyelv közös tulajdonsága, hogy számítógépek vezérlésére alkalmas. Ezt a vezérlést nem közvetlenül a gépnek szóló utasításokkal végzik, hanem olyan nyelvet használnak, amely könnyebben megtanulható, értelmezhető, átlátható az ember számára, mint a közvetlen gépikód, ugyanakkor az ilyen nyelven írt program egyértelműen lefordítható gépi utasítások sorozatára. Ezt az átalakítást szintén programok végzik, amelynek két alapvető típusa ismert, az interpreter vagy értelmező és a compiler vagy fordítóprogram.

Az **interpreter** a programozási nyelv utasításait az eredeti forráskódban, vagy kissé átkódolt és tömörített formában sorra veszi és értelmezi, majd azonnal végrehajtja azokat. Ennek lényeges előnyei és hátrányai vannak a fordításhoz képest. Előny, hogy a program forrásnyelven rendelkezésre áll, bármikor módosítható és a módosított program azonnal kipróbálható. A jobb interpreterek még a forrásnyelvű utasítások nyomkövetését is megengedik, a programozó az általa írt utasítások hatását közvetlenül megvizsgálhatja. Lényeges hátránya viszont az interpretereknek viszonylagos lassuságuk. Ha egy utasítást az interpreter egy program futása során egymilliószor hajt végre, akkor egymilliószor értelmezi jelentését, elemzi helyességét és csak azután tudja végrehajtani.

A **fordítóprogram** a forrásnyelven írt utasításoknak közvetlenül végrehajtható gépiódu programot feleltet meg, amely majdnem (!) olyan hatékony, mintha gépióduban írtuk volna. Nagy előnye ennek az eljárásnak, hogy a lefordított program önmagában képes működni és meglehetősen gyors. Hátránya, hogy módosítása a fordítási eljárás megismétlését vonja maga után, ami különösen mikroszámítógépen igen időigényes. Ezenkívül a lefordított program nehezen nyomkövethető, hiszen az eredeti nyelvi elemek már javarészt nem figyelhetők meg, különíthetők el benne.

Mindezekből következően az igazán jó nyelvi rendszerek ugyanannak a programozási nyelvnek mind interpreterét, mind fordítóprogramját tartalmazzák, sőt a programtesztelést is segítik az eredeti nyelvi elemeket használó nyomkövetési eszközökkel. Sajnos ilyen rendszerre a mikroszámítógépek területén alig van példa.

A programozási nyelvek egy lehetséges felosztása aszerint osztályozza a nyelveket, hogy mennyire kell az emberi nyelven megfogalmazott vezérlési utasítást - programot - átdolgozni, hogy az adott nyelven helyes, az eredeti feladatot megoldó programot kapjunk. Eszerint beszélünk:

- assembly nyelvekről,
- géporientált nyelvekről (szerencsére erre nemigen van példa mikroszámítógépen),
- magasszintű nyelvekről (pl. BASIC, FORTRAN, PASCAL, ADA),
- nagyon magasszintű vagy logikai programozási nyelvekről (pl. PROLOG, LISP, MODULA),
- ujtípusú programozási nyelvekről (pl. FORTH, C).

A legutolsó csoport meglehetősen furcsán hangzó elnevezése és az új csoport maga azért keletkezett, mert ezeket a nyelveket nehéz nyugodt lélekkel besorolni az előző négy osztályba.

Az **assembly nyelvek** (amelynek egy képviselőjével fogunk ebben a könyvben foglalkozni) mindig egy adott processzor-család utasításkészletéhez kötődnek, és csupán a gépi utasítások egyszerűbb, könnyen megjegyezhető írásmódját, valamint nem túl bonyolult kifejezések használatát teszik lehetővé. Az assembly nyelvre (a makrólehetőségektől eltekintve) jellemző az egy assembly utasítás - egy gépi utasítás megfeleltetés.

A **géporientált nyelvek** szintén egy hardware-hez kötődnek, de nyelvi elemeik, utasításaik már több gépiódu utasításnak felelnek meg. A gépek közel olyan hatékony vezérlését teszik lehetővé, mint az assembly programozással elérhető.



A **magasszintű nyelvek** legfontosabb jellemzője, hogy gépfüggetlenek (többé-kevésbé) és aránylag bonyolult feladatok írhatók le egy-egy utasításukkal. Ezért le kell mondani a csak konkrét hardware-re jellemző sajátosságok kihasználásáról és a keletkező (fordított vagy interpretált) gépikód hatékonysága elmarad az assembly programétól. Ezek a nyelvek algoritmikusak, ami azt jelenti, hogy a program a megoldás pontos mikéntjét írja le, ha aránylag magas szintű utasításokkal is.

A nagyon magas szintű vagy **logikai programozási nyelveken** – az utóbbi inkább jellemző megnevezés – irt programok nem a megoldás módszerét, hanem inkább a feladatot magát fogalmazzák meg, azután a feladatot egy, vagy néhány algoritmustípussal oldják meg. Jellemző példája ennek a PROLOG. Itt a gyors programozhatóságért, az igen jól olvasható programért általában csökkent hatékonysággal kell fizetni. Ugyanekkor ezek a nyelvek alkalmasak olyan bonyolult feladatok megfogalmazására is, amelyeket alacsonyabb szinten emberi korlátokból fakadóan lehetetlen vagy csak igen nagy ráfordításokkal volna lehetséges megoldani. Ilyenek az ún. szakértői rendszerek, amelyeket – néha egymásnak teljesen ellentmondó definíciókkal – ma sokat emlegetnek.

Az **ujtípusú nyelvek** egyrészt a magasszintű nyelvekhez hasonlítanak utasításaik és programszerkezetük bonyolultságában, másrészt az assembly nyelvhez az elérhető gépi lehetőségekben. Tulajdonképpen ezek az igazán géporientált nyelvek, de nem egy bizonyos géptípusra, hanem általában a számítógépekre – legalábbis a mai, ún. negyedik generációs gépekre – orientáltak. A mikroszámítógépek esetében igen ígéretesek a FORTH nyelv által nyújtott lehetőségek.

A fenti igen rövid eszmefuttatásból is látszik, hogy igazán univerzális nyelvi eszközzel ma még nemigen rendelkezünk. Ezért egy adott feladat megoldásának lényeges eleme a megfelelő nyelv kiválasztása. Ennek során egymásnak ellentmondó szempontok közös optimumát kell megtalálni, hogy az adott gépen rendelkezésre álló nyelvi eszközök közül melyikkel oldjuk meg az adott feladatot aránylag a legkisebb ráfordítással és a legnagyobb hatékonysággal.

Ugyanakkor azt látjuk, hogy ma csaknem minden feladat mikroszámítógépes megoldására BASIC-et használnak, mint univerzális eszközt. A BASIC, amely talán a legalacsonyabb szintű "magasszintű" nyelv, valóban aránylag könnyen megtanulható, segítségével gyorsan lehet közepesen bonyolult feladatokat elfogadható módon megoldani. Ezenkívül egyetlen előnye van, elterjedtsége (igaz, hogy az egyes gépeken használt BASIC nyelvek néha drámaian eltérnek egymástól).

Számos hátránya van. Ahelyett, hogy részletesen elemeznénk ~~ezeket~~ ezeket, elég itt annyi, hogy mielőtt elkezdnék egy adott feladatot megoldani, vizsgáljuk meg azért a rendelkezésünkre álló többi lehetőséget is. Persze, akárhogy szidjuk is, a BASIC olyannyira elterjedt, hogy még hosszú ideig a mikroszámítógépek, különösen a hobby-kategória "szabványos" nyelve lesz.

A másik véglet, amivel szintén gyakran találkozunk az, hogy a BASIC nyilvánvalóan nem alkalmas ennek és/vagy bármilyen feladatnak a tisztességes megoldására, ezért a programot assemblyben kell megírni. A következtetés két ponton is hibás. Először is nem mindig igaz, hogy a BASIC alkalmatlan az adott és különösen nem, hogy bármilyen feladatra. Másodszor, ha az adott esetben a BASIC alkalmatlan is, még nem biztos, hogy rögtön a legnagyobb ráfordítást igénylő eszközzel kell a feladatnak nekiesni. Még az sem igaz, hogy minden feladatot meg lehet oldani assemblyben, mivel egy adott szerkezeti bonyolultság felett az assembly program tökéletes tesztelése szinte lehetetlen. Ha mégis tisztességesen meg akarjuk oldani, annyira szét kell a feladatot kisebb részekre bontani, hogy ezzel szinte új programozási nyelvet alkotunk, ezáltal veszünk a hatékonyságból is.

Ezután foglaljuk össze nagyon röviden, hogy milyen programozási nyelveket valósítottak meg a **Commodore 64**-en.

Az **assembly** nyelvről nem szolunk itt, mivel ezzel foglalkozik az egész könyv hátralévő része.

A **géporientált nyelvek** szerencsére elkerülték a Commodore 64-et és nem adtak hozzá egy speciális és másutt alkalmazhatatlan eszközt.

A **magasszintű nyelvek** közül az első természetesen a **BASIC**. A beépített BASIC interpreterrel nem lehet különösebben dicsekedni, bár amit tud, azt legalább többé-kevésbé hibamentesen tudja.

Bánsó hiányosságai a PRINT USING, az ELSE, az ERASE, a tisztességes CHAIN hiánya. A nem kis számú BASIC kiterjesztések (a legismertebbek a SIMON's BASIC és a GRAPHIC BASIC) is sajnos nem a már-már szabványosnak tekinthető MICROSOFT BASIC, vagy a szintén nem rossz nagygépes BASIC4+ (DEC) irányába terjesztik ki a nyelvet, hanem a Commodore speciális lehetőségeit teszik kényelmesen elérhetővé.

Említésre érdemes a C64-en elérhető nyelvek közül a **PILOT** és **LOGO**, amelyek főleg gyermekek oktatásában hasznosak és a **COMAL**, amelyet a Commodore cég a BASIC helyett ajánl, de nálunk alig ismert.

A többi magas szintű nyelv közül használható formában egyedül a **PASCAL** érhető el Commodore-n. Itt nem a Data Becker PASCAL 64 fordítóprogramjára gondolunk, ami a PASCAL nyelv és általában a fordítóprogramírás paródiája, hanem inkább az ügyes ZOOM PASCAL-ra, és méginkább az igazán komoly Oxford-Pascal-ra. Gondot jelent azonban, hogy az ezek által elfogadott nyelvek meglehetősen különböznek egymástól.

A Commodore 64-es **ADA** és **microProlog** fordító ill. interpreter inkább a nyelvek - felületes - bemutatására, mint feladatmegoldásra való.

Az újtipusú nyelvek közül a **FORTH** megfelelő módon használható interpreterrel - sőt több, kisebb-nagyobb mértékben eltérő interpreterrel - rendelkezik Commodore 64-re, bár ezen a gépen való sikereiről még nem sok hír érkezett.

Ezen rövid bevezető után, amellyel reméljük nem untattuk az olvasót, térjünk rá könyvünk tárgyára, a Commodore 64 assembly programozásához szükséges ismeretekre.

## 1.2 Utmutató a 2.-5. fejezethez

Bevezetésül, szeretnénk eligazítani az olvasót a további fejezetek felépítésében és rendeltetésében, és néhány tanácsot nyújtani a könyv használatához, valamint egy kicsit foglalkozni az alkalmazott terminológiával.

A 2. fejezet a MOS 6510-es mikroprocesszor működésével ismerteti meg. Igyekeztünk azokkal és csak azokkal az ismeretekkel foglalkozni, amelyek elengedhetetlenül szükségesek az alapszintű assembly programoz elsajátításához.

A 3. fejezetben a processzor utasítás-készletével foglalkozunk, részletesen tárgyalva valamennyi utasítást.

A 4. fejezet majdnem kézikönyv-szerűen mutatja az assembly programozásban felhasználható legismertebb software-eszközöket. Célunk vele az, hogy az eredeti kézikönyvvel nem rendelkező vagy annak nyelvét nem értő olvasó az itt leírtak alapján képes legyen ezen eszközök legalább alapfoku szintű alkalmazására. Igyekeztünk a Magyarországon leginkább elterjedt eszközöket összegyűjteni, erőnyeiket és hibáikat elemezni.

Az 5. fejezet, amely mind közül a legterjedelmesebb, az assembly programozás módszereibe enged bepillantást, és megpróbál enyhíteni a kezdő programozó súlyos gondjain, egy-ézersmind ötleteket adni a helyes assembly-alkalmazáshoz.

A függelékben néhány - talán hasznos - táblázattal segítjük a gyors tájékozódást egy-egy probléma esetén.

Külön felhívjuk a figyelmet az irodalom-jegyzékre, amely a további - sajnos többnyire nyelvtudást igénylő - tájékozódáshoz igyekszik segítséget nyújtani.

A könyvben tárgyaltak elsajátítását nagymértékben elősegíti a könyvvel együtt, de külön is megvásárolható lemez, amely az összes teljes programot és nagyobb lélegzetű szubrutint tartalmazza, természetesen Commodore 64 VC 1541-es formátumban. A lemezen többféle program is található:

- Az /ASS-re végződő nevűek az assembly nyelvű forrásprogramot tartalmazzák HELP+, azaz BASIC interpreter-kód formában. Így minden segédeszköz nélkül listázhatók, módosíthatók.
- Az /MC jelűek az azonos nevű /ASS program gépikódra fordított változatai, amelyek LOAD "<programnév>",8,1 utasítással tölthetők a memóriába. Így a példákat azok az olvasók is kipróbálhatják, akiknek nincs assemblerük.
- A /BAS névvégződésű file-ok az assembly programnak megfelelő BASIC programot tartalmazzák - csak a bonyolultabb programok esetén.

- A /TESZT jelu file-ok a megfelelő program tesztelését végzik, segítik elő, szintén nem minden programhoz van ilyen.
- A lemezen lévő egyetlen /TRACE névvégződésű - BASIC - file olyan speciális /TESZT program, amely a működést nyomozó változó kiírásaival - reméljük - közelebb visz a bonyolult algoritmus megértéséhez.

Nagyon nehéz egy szakkönyv - s reméljük munkánk annak mondható - pontos olvasótáborát kiválasztani, másként fogalmazva, olyan könyvet írni, amely aránylag széles réteg igényeit elégíti ki. Ez a könyv azzal a szándékkal íródott, hogy a Commodore 64-et ismerő, BASIC programozásban jártas, de nem túl sok assembly ismerettel rendelkező érdeklődő amatőr vagy kezdő hivatásos programozó igényeinek megfelelően, számukra olyan információkat nyújtson, amelyek segítségével önállóan képesek lesznek egyszerűbb assembly programozási feladatok megoldására.

Végezetül a könyv **terminológiájáról**. Napjainkban elég sok vita dúl az idegen és különösen a számítástechnikai szakmai kifejezések használatáról, írásmódjáról. Ebben a munkában nem kerültük görcsösen az idegen - jobbára angol - kifejezések használatát, hiszen ezek meglehetősen elterjedtek, ugyanakkor eléggé közismert jelentésűek. Az idegen kifejezéseket - kivéve a teljesen meghonosodottakat - mindenütt az eredeti írásmód szerint írtuk, ami szintén vitatható, de legalább korrekt és következetes eljárás.



## 2. Mit tud a 6510-es mikroprocesszor. Programozási modell

Ebben a fejezetben azt az eszközt szeretnénk bemutatni, amelynek vezérlése fő feladatunk, a MOS Technology **6510-es jelű mikroprocesszorát**. A 6510-es nem az egyetlen integrált áramkör a Commodore 64-ben, de kétségkívül a legfontosabb, hiszen a központi egység szerepét tölti be, ha a régebbi számítógép-fogalmakkal akarjuk megmagyarázni szerepét. Mint később utalni fogunk rá, az olvasó bizonyára jól tudja, a gépben (és a perifériákban) található egyéb chippek input/output vezérlők szerepét töltik be – bár néha igen-igen kiterjesztett értelemben (lásd pl. a VC-1541 lemezegység vezérlőjét, amely egyébként egy MOS 6502-es mikroprocesszor).

A 6510-es mikroprocesszor-chip a MOS Technology korábbi 6502-es gyártmányának kismértékben módosított, továbbfejlesztett változata. Megjegyzendő, hogy a két processzor utasításkészlete és az utasítások hatása teljesen megegyezik.

A 6502-es a Commodore VIC 20 és PET gépének, valamint az Apple II sorozatnak lelke többek között. Ezért az ebben a könyvben leírtak a speciálisan Commodore 64-re jellemző részletek elhagyásával az említett gépek programozásához is segítséget nyújthatnak, bár erre a különválasztásra sehol sem utalunk, mivel feladatunk a Commodore 64 programozásának támogatása.

A 6510-es leírásában kizárólag a programozás szempontjából lényeges jellemzőket fogjuk tárgyalni, **kerülve minden**, ebben a tárgyalásban **felesleges hardware-részletet**. Ezért szerepel a fejezet címében a "programozási modell" kifejezés, mivel itt a mikroprocesszor egy "software kép"-ét adjuk. Ennek során leírjuk azt, hogy mik a 6510-es komponensei és hogyan tartanak kapcsolatot egymással a működés során.

### 2.1 Néhány alapfogalom

A 6510-es egy olyan processzor, amely adatokat alakít át a tárolt program utasításainak engedelmessé. Mind az adatok, mind a programutasítások ún. gépi memóriában tárolódnak, amelyek fizikailag integrált áramkörökkel vannak megvalósítva a C64-en belül. Ennek a tárolónak két alapvetően különböző típusával kerülünk majd szembe, a **ROM**-mal és a **RAM**-mal. A ROM (Read Only Memory) csak olvasható memória, amelybe a C64 gyártása során információt rögzítettek (többnyire programutasításokat, bár nemcsak azt). A RAM (Random Access Memory) azaz félrevezető neve szerint véletlen elérésű tároló a fentivel ellentétben üres, de bármikor átírható, megváltoztatható a tartalma, tetszésünk szerint. Ez a memóriatípus szolgál arra, hogy programjainkat és adatainkat benne tároljuk.

A memória byte-okra, azaz 8 bináris jegyből álló részekre oszlik, amelynek mindegyike egy külön sorszámmal, címmel rendelkezik. Egy byte-on  $2^8$  különböző számkombináció ábrázolható, amelyeket tetszés szerinti módon értelmezhetünk. Az egyik szokásos értelmezés szerint ezek 0 és 255 közötti számokat jelölnek, egy másik szerint a karakterek (betűk, számjegyek, jelek) kódjait, ismét egy másik szerint a különböző kódok különböző gépi utasításokat jelentenek. Hogy egy adott byte-ot, annak tartalmát hogyan kell értelmezni, azt mindig az értelmezőnek (a programnak, illetve a programozónak) kell tudni, azt külön semmi sem jelöli.

A 6510-es mikroprocesszor természetesen össze van kapcsolva a **memóriaegységekkel**. Ezt a kapcsolatot **adatbusznak** nevezzük. Ezen a nyolc huzalon keresztül lehetősége van a mikroprocesszornak arra, hogy megváltoztasson egy memóriabyte-ot, illetve kiolvassa tartalmát. Hogy most éppen írni, vagy olvasni kíván-e, azt egy másik adatkapcsolat, a vezérlőbusz egyik bitjével jelzi. A többi bitek egyéb, a külvilággal való kapcsolatokat vezérelnek. A megfelelő byte kiválasztása a memóriából természetesen a címével (sorszámaival) történik. Mivel a feladatok megoldásához meglehetősen nagyszámú programutasításra és adatra van szükség, a címek megjelölésére nem 8, hanem 16 bináris jegyből álló számokat használunk, a megadásukra szolgáló busz 16 párhuzamos elektromos kapcsolatot jelent a központi mikroprocesszor és a memória között. Ilyen módon  $2^{16}$  azaz 65536 cím és a neki megfelelő byte különböztethető, címezhető meg. A címeket és többnyire a memóriabyte-ok tartalmát is hexadecimális számokkal szokás jelölni, amelyeket a 6510-es assembler konvencióinak megfelelően a \$ jellel fogunk mindig bevezetni. Így egy memóriacím mindig legfeljebb 4-jegyű hexadecimális szám \$0000-tól \$FFFF-ig, egy byte értéke pedig \$00 és \$FF közé esik, a határokat is beleértve.

Ebben a munkában nem foglalkozunk olyan elemi ismeretekkel, amelyek a hexadecimális és bináris számábrázolással kapcsolatosak. Tisztázni kívánunk viszont néhány elemi fogalmat az egységes nomenklatura kedvéért. Egy **byte** legbaloldali bitje (a byte 7. bitjének is szokás nevezni) a szám előjelét hordozza, ha az ábrázolt értéket -127 és +127 közötti számként értelmezzük. Ha a 7. bit magas, a számot kettes komplementens ábrázolásu negatív számnak tekintjük. Két egybyte-os szám összeadásával olyan számot kaphatunk, amelynek értéke nagyobb 255-nél, így egy byte-on már nem ábrázolható. A keletkezett szám jobboldali 8 bitjét nevezzük a művelet eredményének, míg a legmagasabb helyiértékű bitet átvitelnek (carry-nek).

Ha 255-nél nagyobb számokat kívánunk alkalmazni, természetesen több byte-ot kell együtt kezelnünk. Ennek leggyakoribb esete a **címek** ábrázolása, amelyre mint tudjuk, a Commodore 64 esetében 16 bitre, két byte-ra van szükségünk. A kétbyte-os

egészeket a 6510-es alacsony byte - magas byte sorrendben kezeli a memóriában. Ez azt jelenti, hogy a szám jobboldali két hexadecimális jegye van a kisebb és a baloldali kettő a nagyobb memóriacimen.

Lássunk erre egy kis BASIC nyelvű példát:

Tegyük fel, hogy a memória \$0014-es címén egy kétbyte-os cím van (tehát a \$14 és \$15 címeken, ahol a \$14 a kisebb és \$15 tartalmazza a nagyobb helyiértékű byte-ot).

A következő BASIC sor ezt a címet nyomtatja ki decimálisan:

```
KH=PEEK(20):NH=PEEK(21):PRINT 256*NH+KH
```

A 20 és 21 természetesen \$14-et, illetve \$15-öt jelent, mivel a BASIC nyelvben csak decimális értékek írhatók.

## 2.2 Regiszterek és jelzők

A 6510-es mikroprocesszor az adatokon bizonyos előre meghatározott utasításokat képes végrehajtani. Az **utasítások** egy, két vagy három byte-osak és mindig az első byte tartalmazza az utasításra utaló kódot. Az utasítások végrehajtásához a mikroprocesszor maga is tartalmaz hét byte-nyi adattárolót, amelyeket **regisztereknek** nevezünk és mindegyikük speciális célokat szolgál. Az utasítások segítségével adatot cserélhetünk a memória és a regiszterek között, valamint logikai és aritmetikai műveleteket végezhetünk ezekkel a tárolt adatokkal. A mikroprocesszor működése programozási szempontból abból áll, hogy a memóriában tárolt programot utasításról-utasításra végrehajtja.

A 6510-es **regiszterei** közül az első kétbyte-os, ez a **programszámláló**, rövidítve PC (Program Counter). Azért van két byte-ra szükségünk, mert a 6510-es által elérhető 64 Kbyte-nyi tár egy címét kell benne tárolni, mégpedig mindig éppen a soron következő utasítás címét. Hogy melyik a soronkövetkező utasítás, azt természetesen a mikroprocesszor maga határozza meg a működés során, attól függően, hogy éppen milyen utasítást hajt végre. Általában ez az éppen végrehajtott utasítást a memóriában fizikailag követő utasítás, de bizonyos utasítások, vagy külső állapotok változásai ezt megváltoztathatják, mint a későbbiekben, az utasítások részletes tárgyalásánál látni fogjuk.

Amikor a Commodore 64 gépet bekapcsoljuk és a processzorra jutó feszültség értéke eléri az 4,75 V-ot, egy ún. **inicializációs folyamat** játszódik le, amelynek során a \$FFFC-FFFF címen lévő két byte értéke a programszámlálóba töltődik, azaz ez lesz a bekapcsolás után végrehajtott első utasítás címe. Az ilyen, a memóriában tárolt kétbyte-os programcímek vektornak szokás nevezni. A fent leírtakból következik, hogy bekapcsoláskor a memória \$FFFC címen egy programcímnek, tehát értékes információnak kell lenni, ami csak úgy valósítható meg, ha mind ez a memóriacím, mind az, ahova ez a vektor mutat, ROM-ban van.

A következő fontos regiszter az **akkumulátor**, rövidítve A. Szoktuk aritmetikai regiszternek is nevezni, mivel a legtöbb aritmetikai (és logikai) utasításban résztvesz. Ezeknek az utasításoknak a segítségével az akkumulátorhoz egybyte-os értékeket lehet hozzáadni vagy kivonni belőle és az eredmény mindig az akkumulátorban képződik. Ezt az új tartalmat azután a memória tetszőleges címén tárolhatjuk. Az akkumulátor egybyte-os regiszter.

Az **indexregiszterek** (X és Y regiszterek) alapvetően két célra szolgálnak, bár ez a két feladat gyakran összeolvad. Egyrészt egyszerű utasításokkal egyesével növeljük vagy csökkentjük tartalmukat, így számlálásra használhatjuk őket.

Másik funkciójuk az utasításokban foglalt memóriacímek módosítása, amelyek segítségével bonyolult címzésmodok valósíthatók meg. Ezek a nyolcbites regiszterek szintén a memóriából tölthetők, illetve értékek egy-egy memóriabyte-on tárolható.

A memória második 256 byte-ját (az első lapot) tehát a \$0100-tól \$01FF-ig terjedő címtartományt a 6510-es veremként használja. A verem egy speciális tömb, amelyet bizonyos utasítások automatikusan használnak. Itt tárolódnak a szubrutinok, illetve a megszakítások visszatérési címei. A szubrutinok olyan alprogramok, programrészek, amelyek a program több helyéről aktivizálhatók, s a hívás helye öröködik meg a veremben, amelynek segítségével talál vissza a program a szubrutinhívást követő utasításra. A verem soronkövetkező szabad helyét (a lapon belül) tárolja egy újabb regiszter, a veremmutató angol nevének rövidítésével SP (Stack Pointer). A veremmutató kezdeti értéke bekapcsolás után \$FF lesz, ami mint cím \$01FF-ként értelmezhető. Mivel a felső byte mindig \$01, ennek tárolására nincs is szükség, így a veremmutató is egybyte-os regiszter. Az első szubrutinhívás (JSR utasítás) hatására a veremben tárolódik a visszatérési cím (két byte-on) és a veremmutató értéke kettővel (\$FD-re) csökken. A szubrutin újabb szubrutint hívhat, ami újabb címet ment el a verembe és ez a verem méretéből következően legfeljebb 128-szor ismétélhető. Minden szubrutinhívás (logikailag) egy visszatérési (RTS) utasítással kell, hogy végzódjon, amely az utasításszámlálóba tölti a verem tetején lévő két byte-ot és növeli a veremmutató értékét kettővel, ezáltal folytatja a program végrehajtását a szubrutinhívást követő utasítással.

A szubrutinhíváson kívül az ún. **megszakítások** okoznak még automatikus veremműveleteket. A megszakítások a külső világból érkező jelek a mikroprocesszor számára, tehát nem a saját állapotaiból következnek közvetlenül. Két típusát különböztetjük meg a 6510-es esetben, attól függően, hogy azok a mikroprocesszor két megszakításvonalának melyikét aktivizálják, a mikroprocesszor melyik kivezetésének változtatják meg az állapotát. Az egyik, a nem maszkolható megszakítás (NMI - Non Maskable Interrupt) felfüggeszti a program futását, tekintet nélkül arra, hogy milyen állapotban van a processzor. Erre a gép különböző egységeinek pontos időzítése miatt feltétlenül szükség van. Ezzel szemben a második vonalon jelentkező megszakítások a program által kizárhatók, maszkolhatók. Ez a maszkolható megszakítások vonala (IRQ - Interrupt Request). Mindkét típusu megszakítás a verembe menti az utasításszámláló aktuális értékét (hasonlóképpen, mint a szubrutinhívás) és az állapotregiszter (lásd később) értékét is. Ennek megfelelően a veremmutató értékét hárommal csökkenti.



Végso ténykedésként az utasításszámlálóba tölti a megszakításkezelő szubrutin címét tartalmazó vektort. Ez NMI esetén a \$FFFA-\$FFFB, IRQ esetén a \$FFFE-\$FFFF memóriacimekről történik.

Természetesen a megszakításkezelő szubrutinok is egy speciális utasítással (RTI) érnek véget, amely visszaállítja a megszakítás előtti állapotot. Ezt megelőzően újra engedélyezi megszakítást a megszakításjelző (lásd később) magasra állításával. Ez hasonlóan működik, mint a fent leírt RTS utasítás, de az utasításszámlálón kívül az állapotregisztert is vissza menti a veremből.

A fentieken kívül külön utasításokkal is kezelhetjük a vermet, amelyet pl. szubrutinok paraméterátadására használhatunk.

Az utolsó regisztert tulajdonképpen bitenként értelmezzük. Ez az **állapotregiszter**, rövidítve P (Program status register) és egyes bitjeinek külön elnevezése is van, ezek a jelzők.

A P regiszter 7. (legbaloldalibb) bitje a **negatív jelző**, az N. Ennek értékét (mint a többi jelzőét is) bizonyos utasítások megváltoztatják, mások hatástalanok rá. Nevét onnan kapta, hogy értéke magas, ha egy aritmetikai művelet eredménye negatív, azaz legmagasabb helyiértékű bitje magas. Ezt az értéket feltételes programelágaztatásra használhatjuk.

A V jelző, vagy **tulcsordulásjelző** a kettes komplementis tulcsordulást jelzi kivonás esetén. A BIT utasítás szintén kezeli, mint majd ott részletesen tárgyaljuk. Ez a jelző az állapotregiszter 6. bitje. Az 5. bit értéke közömbös programozási szempontból.

A B vagy **software megszakítás jelző** az állapotregiszter 4. bitje. Akkor állítódik magasra, ha az éppen végrehajtott utasítás egy BRK (megszakítás) utasítás. Ennek hatása ugyanolyan, mintha egy maszkolható megszakításkérelem érkezett volna, így jól használható programok kiprobálása során. Az egyetlen dolog, amiből a megszakításkezelő szubrutin megtudhatja, hogy software-megszakítás történt, az a B jelző. Jó megjegyezni, hogy bár a BRK a maszkolható megszakítást váltja ki, hatása nem maszkolható, ezért hibás (BRK-ra futó) megszakításkezelő szubrutinok könnyen végtelen megszakításba torkolhatnak.

A D jelző, az állapotregiszter 3. bitje a binárisan kódolt **decimális üzemmódot** jelzi, ha értéke 1. Ez az összeadás (ADC) és kivonás (SBC) utasításra érvényes. Ebben az esetben (decimális módban) egy byte két, négy biten ábrázolt decimális számjegyből áll, az ábrázolható érték tehát a 0-99 tartományba esik. A decimális mód beállítására és megszüntetésére külön utasítások szolgálnak.

A 2. bit az I, neve **megszakítás maszkolás jelző**. Ha értéke 1, amit külön utasítással (SEI) érhetünk el, a megszakítások maszkolva lesznek, hatástalanok. Ekkor csak a nem maszkolható (NMI) és software (BRK) megszakítás hajtódik végre. Törölését a CLI utasítás végzi, amely után a program ismét megszakítható.

A Z vagy **nulla jelző** szerepe az N-éhez hasonló, de nem a negatív, hanem a nulla eredmény állítja magasra. Egyébként az állapotregiszter 1. bitje.

A legkisebb helyiértékű, 0. bit az **átvitel (carry) jelző**, a C. Sok utasítás kezeli, nevét az aritmetikai utasításokban betöltött szerepéről nyerte. A feltételes programelágazás gyakori eszköze a Z jelzővel együtt.

A regisztereken kívül a memória első két byte-ja is (a \$0000 és \$0001 című) kitüntetett szerepet játszik, tulajdonképpen regiszterként viselkedik. A 0. byte-ot adatirány-regiszternek, az 1.-et input/output kapunak szokás nevezni. A \$0001 című I/O kapu hat alsó bitje a 6510-es mikroprocesszor-chip egy-egy kivezetésének felel meg. A \$0000 című adatirány-regiszter egy-egy bitje mondja meg, hogy a neki megfelelő bitet az I/O kapuban olvassa vagy írja a processzor (1=olvasás, 0=írás). Ennek segítségével a hat kivezetésen át a processzor kapcsolatot tud tartani a perifériavezérlőkkel és egyéb külső eszközökkel. A perifériális eszközökkel való kapcsolattartás bonyolult folyamat, és nem is csak az I/O kapun át folyik, de vezérlésében a ROM-ban lévő KERNAL szubrutinok is segítenek, így ez elég egyszerűen végezhető, mint majd később látni fogjuk.

csak a 16. és 17. bit

## 2.3 Címzés módok

Az egyes utasítások a következő feladatok valamelyikét hajtják végre:

- adatot töltenek a memóriából valamelyik regiszterbe,
- az előző fordítottjaként valamely regiszter tartalmát egy memóriabyte-ba tárolják, vagy más módon módosítanak egy memóriabyte-ot,
- egyik regiszterből egy másikba töltenek értéket,
- megváltoztatják valamely regiszter állapotát,
- megváltoztatják az utasításslámláló értékét, azaz programelágazást hoznak létre.

Az utasítások első byte-ja, mint már említettük, mindig az utasításkód, amely nemcsak magát a feladatot adja meg, hanem a címzés módot is, azt hogy a kiegészítő információt (többnyire címet) hogyan kell meghatározni. A címzés mód ezenfelül azt is megadja, hogy az utasításhoz még hány byte információ kapcsolódik, kettő, egy vagy egy sem. A 6510-es mikroprocesszor igen változatos címzés módokkal rendelkezik, amelyeket a következőkben részletesen tárgyalunk.

Az **abszolút címzéshez** hárombyte-os utasításhossz tartozik, tehát a cím 2 byte-os. Ekkor az utasításkódot követő két byte a cím maga, kisebb helyiértékű byte - nagyobb helyiértékű byte sorrendben. Ilyenkor tehát az utasítás maga tartalmazza a címet, amelyet a cím-buszon el kell helyezni. Ha pl. a soronkövetkező utasítás így fest:

\$AD,\$01,\$0B

a \$AD utasításkód azt jelenti, hogy egy byte-ot a memóriából az akkumulátorba kell tölteni (LDA) abszolút címzés móddal, tehát a címet a második és harmadik byte adja meg. Ezt az utasítást assembly nyelven így íránk:

LDA \$0B01

Az utasításkódot az assembly nyelvben a könnyebben megjegyezhető mnemonikkal (LDA - Load Accumulator) helyettesítettük, a cím a jobban olvasható természetes sorrendű, a címzés módot nem kell külön jelölni, ez itt abszolút címzést jelent. Az assembler ezt az utasítást éppen az előző alakra, a gépi megfelelőre konvertálja.

A **nullás lapu címzés** a 6510-es mikroprocesszor azon speciális tulajdonságát használja ki, hogy az a memória első 256 byte-ját (\$0000-\$00FF memóriacímek) kitüntetetten kezeli. Az utasításkódok egy külön csoportja nullás lapu címzést jelöl, ekkor az utasítás csak két byte hosszú, az operandus csak a cím második (kiseb helyiértékű) byte-ját tartalmazza, mivel az első byte mindig \$00. Így az utasítás rövidebb lesz és a memória elérése is kevesebb időt igényel. A legtöbb

assembler a nullás lapu címzésnél sem kíván külön jelölést, hiszen a cím nagyságából maga is ki tudja számolni, van-e lehetőség a rövidebb címzés mód alkalmazására. Mint már említettük, a nullás lap mellett az egyes lap is kiemelt szerepet játszik, mint verem-memória.

(pl. MOV, CLD)  
A közvetlen címzés mód esetén szintén csak két byte-ra van szükség az utasítás megadására. Ebben az esetben a második byte nem cím, hanem magát az operandust (egy konstans értéket) tartalmazza, ilyenkor nem kell külön a memóriához fordulni a processzornak.

Pl. a

LDA #\$FF

(assembly) utasítás a \$FF értéket tölti az akkumulátorba. A közvetlen címzés assembly jele a #. Az utasítás gépi kódú megfelelője a

\$A9,\$FF,

egymást követő két byte-on elhelyezve látjuk, hogy az LDA (Load Accumulator) utasítás kódja közvetlen címzés mód esetén \$A9, míg abszolút címzésnél \$AD volt. Ez, mint már említettük, azért van így, mert az utasításkód nemcsak az utasításra magára, hanem a címzés módra is utal. Az utasítás és a címzés mód együtt már egyértelműen meghatározza az utasításkódot, és megfordítva.

(pl.: LR: R R regiszter, az operandus a regiszterben van)  
A beágyazott címzés mód esetén az utasítás egyetlen byte-ból, magából az utasításkódból áll. A legrövidebb alaknak megfelelően az ilyen címzés módú utasítások végrehajtási ideje a legrövidebb. Mivel nincs lehetőség operandus megadására, ezek az utasítások mindig csak a 6510-es regisztereivel és jelzőivel és esetleg a veremmel vannak összefüggésben. Egyszerű példa regiszterből regiszterbe való átvitelek esete, mint pl. a

TAX

assembly utasítás, amely az akkumulátor tartalmát az X regiszterbe viszi, de vannak bonyolultabb funkciójú utasítások is, mint pl. az

RTI

megszakításból való visszatérési utasítás, melynek több feladata is van.

Ebben a könyvben egy kissé eltérünk a 6510-es gépi kódú programozását tárgyaló munkák gyakorlatától. Ezekben egy - itt nem használt - további címzés módot is tárgyalnak, az ún. akkumulátoros címzést. Az akkumulátoros címzést a léptető utasítások (LSR, ASL, ROL, ROR) esetében lehet alkalmazni, ha az operandus az akkumulátor. Mivel ez esetben az utasítás egy

byte hosszú és minden tekintetben megfelel a beleértett címzésmod jellemzőinek, a továbbiakban akkumulátoros címzésmod helyett mindig beleértett címzésmodot fogunk használni. Meg kell ugyanakkor említeni azt, hogy a fenti utasítások beleértett címzésmodu változatánál az assemblerek megkívánják az A fiktív operandus kiírását, (pl. ASL A), amelyet ezért tiltott címkeként használni.

**A relatív címzésmodot** a feltételes vezérlésátado utasítások használják. Ezeknek a segítségével a program bizonyos feltételek teljesülése esetén nem a fizikailag soronkövetkező, hanem egy, az utasítás operandusa által megcímzett utasítással folytatódik. Ezekről az utasításokról a későbbiekben részletesebben is szólnunk, most a címzésmod bemutatásához ismerjünk meg csak egyet, a BEQ (Branch on Equal) utasítást, amely akkor okoz vezérlésátadást, ha a nulla jelző magas. Ez általában akkor következik be, ha az akkumulátort nulla értékkel töltöttük, vagy ha egy összehasonlítás eredménye egyenlőségre utal. A következő kis programrészletben az akkumulátorba a nullás lap \$14 című byte-jának tartalmát töltjük, majd megnöveljük eggyel az Y regiszter tartalmát (INY - Increment Y register), ha a \$14 tartalma nem nulla. Az assembly nyelvű utasítássorok előtt az utasítás kezdőcíme, és az utasítás hexadecimális kódja szerepel byte-onként:

```

C100  A5 14      LDA $14
C102  F0 01      BEQ N
C104  C8         INY
C105  A5 15  N  LDA $15
          .
          .
          .

```

A fenti formátum egyébként az assembler lista alakja.

Ha a \$C100 címen lévő LDA utasítás végrehajtása során a \$14-es cím tartalma nulla, az utasítás egyik következménye az, hogy az SI regiszterben lévő nulla-jelző bit magas lesz. Az ezt követő BEQ utasítás végrehajtása - mivel a feltétel igaz - azt jelenti, hogy az utasításszámlálóhoz hozzáadódik az operandus (előjelesen) így a soronkövetkező utasítást nem a \$C104, hanem a \$C104+1=\$C105 címrol fogja venni a mikroprocesszor. Az assembly írásmódban a vezérlésátadás helyét címkével (N) jeleztük, s a fordítás során az assembler gondoskodik a relatív operandus (esetünkben 1) kiszámításáról. Mivel egy byte értéke, a határokat is beleértve, -128 és +127 közé esik, a legnagyobb relatív ugrás távolsága is csak ekkora lehet, mivel az operandus mindig pontosan egy byte-os. Az utasítások részletes tárgyalásánál majd látni fogjuk, mi a teendő, ha messzebbre kívánunk ugrani egy feltételtől függően.

Hogy a **negatív számok ábrázolását** megértsük, tegyünk egy kis kitérőt. A 6510-es mikroprocesszor (és általában a számítógépek) a negatív számokat ún. kettes komplementes kódban áb-



rázolják. Negatív az a szám, amelynek legbaloldalibb bitje (amit előjelbitnek is szokás nevezni) magas. Most tekintsük az egybyte-os számokat, de az alábbiak igazak a többbyte-os számokra is. Ha egy negatív szám egybyte-os (8 bites) kettes komplement kódú ábrázolását akarjuk megkapni, a számot hozzá kell adni \$100-hoz (decimális 256), az eredmény a keresett ábrázolás lesz. Így a -1 kettes komplement kódja \$100-1=\$FF. Ez a megoldás igen leegyszerűsíti az aritmetikát, hiszen -1+1 számítása

\$FF	vagy binárisan.	11111111
\$01		1
<hr/>		
1/\$00		1/00000000

A keletkezett szám legfelső helyiértékű bitjét (amelyet a byte-on már nem is tudunk ábrázolni) a gép a C (carry) jelzobe viszi, ezt nevezzük átvitelnek. A 8 biten viszont csupán nulla maradt, az eredmény tehát a vártnak megfelelően előállt. Legegyszerűbb, ha megjegyezzük, hogy az egybyte-os pozitív számok \$00 és \$7F közé esnek (0-127), a negatívak pedig \$FF és \$80 közé (-1-től -128-ig).

Az indirekt címzés mód egyetlen utasítás, a JMP - feltétlen ugróutasítás esetén használatos. Az utasítás assembly írásmódjában:

A579 6C 04 03 JMP (\$0304)

a zárójel jelzi az assemblernek, hogy a címzés mód indirekt. Mi a teendő, ha a következő utasítás indirekt ugrás? Az utasításban szereplő kétbyte-os cím egy memóriabyte-ra mutat. A megjelölt és következő címen lévő byte kerül a PC-be, azaz ez lesz a következő utasítás címe. A fenti példában tehát a \$304-es és \$305-ös címen lévő értékek (pl. \$0304 értéke legyen \$7C és \$0305 értéke \$A5) kerülnek a PC-be mint alsó és felső byte (azaz az utasítás: ugrás a \$A57C címre). Az ilyen kétbyte-os, a memóriába lerakott ugráscímeket vektoroknak szokás nevezni. A BASIC interpreter, mint a későbbiekben példát fogunk látni rá, éppen a vektorok segítségével teszi lehetővé a felhasználó gépi kódú programszegmenseinek beszurását az interpreter feldolgozási folyamatába.

Az indexelt címzés módok talán a legbonyolultabbak, ugyanakkor a leghajlékonyabb adatkezelési lehetőséget nyújtják. Az X és Y regisztereket az indexelt címzésre való felhasználhatóságukról indexregisztereknek is szokták nevezni. Az indexelés azt jelenti, hogy egy adatbyte címet nem közvetlenül, hanem két mennyiség összegeként írjuk le. Az elsőt bázisnak, vagy kezdőcímnak, a másodikat pedig indexnek hívjuk, amely megadja, hogy a kezdőcímtől hányadik byte lett megcímezve. Ez ugyanaz a mechanizmus, mint amikor egy egyméterű tömb n-edik eleméről beszélünk, ahol n az elem indexe. A gépi kódú utasítás megadja a kezdőcímet és az indexet. A kezdőcím megadható abszolút címmel, vagy relatív címzéssel, az index pedig az X vagy Y regiszter valamelyike. Ez a címzés-

tipus azért nagyon hasznos, mert az indexregiszter értékének megváltoztatásával ugyanaz az utasítás egy másik byte-ot címez meg, ezt jól ki lehet használni ciklusok szervezésére. Most nézzük az idetartozó három címzésmodot!

**Az abszolút indexelt címzésmodnál** az utasítás egy abszolút címet - a kezdőcímet - tartalmazza, pl.

```
0400          STRING=$400
          .
          .
2000  A2 0A      LDX    #10
2002  BD 00 04   LDA    STRING,X
```

Az első sorban egy címkét definiálunk az assembler számára, azaz egy szimbólumot, amely a programban a \$400 érték helyett fogunk használni. Másszóval a kezdőcím \$400. Az LDX utasítás 10-et tölt (\$0A) az X regiszterbe. Az LDA utasításban ",X" utótag jelzi az assemblernek az abszolút indexelt címzésmodot, ez a gépikódban a \$BD utasításkódban tükröződik, amely az LDA ilyen címzésmodu kódja. Hogyan hajtja végre az utasítást a gép? Először veszi az abszolút címet (\$400), ehhez hozzáadja az X regiszter tartalmát (\$400+\$A=\$40A) és az így kiszámított címen lévő értéket tölti az akkumulátorba. Ha X értéke az utasítás végrehajtásakor történetesen nem \$A, hanem pl. \$30, akkor az A-ba a \$430 című byte tartalma kerül. Az abszolút indexelt címzésmod mind az X, mind az Y regiszterrel használható, és megvan a nullás lapu megfelelőjük is, ami természetesen csak két byte hosszú utasítást eredményez és gyorsabban is hajtódik végre.

**Az indirekt indexelt címzésmod** az indirekt és abszolút indexelt címzésmodok összeolvasztásaként értelmezhető. Ez csak nullás lapu címet tartalmazhat, ennek megfelelően az utasítás csak 2 byte hosszú, pl.

```
A557  91 22      STA    ($22),Y
```

A ",Y" utótag a zárójellel együtt (ami az indirekcióra utal) jelenti az assembler számára az indirekt indexelt címzésmodot. A megjelölt nullás lapu címen (\$22) és soronkövetkező byte-on (\$23) egy kétbyte-os cím található, amely az indirekt címzés kezdőcíme. Az ilyen, általában adattömb kezdetére mutató címet nem vektornak - az a programba mutat -, hanem pointernek (mutatónak) szokás nevezni. Azt mondhatjuk, hogy a \$22, \$23 egy tömb pointerre. Az utasítás végrehajtásának megértéséhez tegyük fel, hogy a \$22 tartalma \$00 és a \$23-é \$C0. Ekkor a kezdőcím a \$22, mint pointer értéke (\$C000) plusz az Y regiszter értéke (pl. \$02), azaz példánkban \$C002, erre a címre kerül az A regiszter tartalma. Ez a címzésmod a leghajlékonyabb az összes között, igen jól alkalmazható.

Az indexelt indirekt címzés mód első pillantásra igen hasonló az előzőhöz, pedig igen fontos, hogy a különbséget lássuk. Ez nemcsak abból áll, hogy a fenti címzés mód csak az Y, ez pedig csak az X regiszterrel használatos. Az assembly írásmód

```
2000 A1 20 LDA ($20,X)
```

már jelzi, hogy itt előbb az indexelést kell végrehajtani és csak azután az indirekciót. Mit jelent ez a gyakorlatban? Tegyük fel, hogy az utasítás végrehajtásakor X értéke 2, míg a memória a következő értékeket tartalmazza:

CIM	ERTEK
\$20	\$00
\$21	\$C0
\$22	\$0A
\$23	\$C0
.	.

Végrehajtáskor a processzor először az utasítás operandusaként megjelölt nullás lapu címhez (\$20) hozzáadja az X értékét ( $20 + \$02 = \$22$ ), majd az ezen pointer által mutatott című (\$C00A) byte értékét tölti az akkumulátorba. Ez a címzés mód talán a legritkábban használt a 6510-es assembly programozása során, mivel segítségével általában egy speciális, ritkán szükséges feladat oldható meg (ráadásul azt is meg lehet másképpen oldani). Tegyük fel, hogy van egy ponttertömbünk a nullás lapon, amely kétbyte-os pointereket tartalmaz. Az X regiszterrel, mint indexszel ilyenkor ki tudjuk választani a tömb elemét és általa a hivatkozott címen lévő byte-ot. Ez igen hasznos lenne ugrótáblák kezeléséhez (CASE programstruktúra), de szerencsétlen módon ez a címzés mód vezérlésátadó utasítások esetén nem használható.

## 2.4 Megszakítások

A megszakítások olyan események, amelyek félbeszakítják a processzor működését, megváltoztatják bizonyos regiszterek tartalmát és a program futását általában új pontról folytatják. A megszakításokat külső (pl. periféria) és belső (órajel, BRK utasítás végrehajtása) tényezők is kiválthatják. Két alapvető típusuk van, maszkolható és nem maszkolható megszakításkérelem. A megszakítás maszkolásán a processzor egy olyan állapotát értjük, amikor a befutó maszkolható megszakításokat figyelmen kívül hagyja. Erre nagy szükség van pl. a megszakítást kezelő szubrutinban, amelynek működése során kellemetlen lenne egy újabb megszakítás, amely újraindítaná a rutint. Tulajdonképpen van egy harmadik megszakítás is, amellyel már találkoztunk; ez a reset, ami a bekapcsolás esetén való indulást, ill. a tisztázatlan helyzetből való újraindulás folyamatát váltja ki. Azért nem szokás ezt megszakításnak nevezni, mert egyrészt nem szakít meg semmilyen tevékenységet, másrészt végrehajtása is különbözik a megszakításokétól, mivel itt a kezdeti állapot elmentése a megszakított tevékenység zavartalan folytatásához hiányzik.

A megszakítások kezelésének első lépése tehát az állapotregiszter (P) és az aktuális cím elmentése a verembe. Ezután az I megfelelő beállítása (ha szükséges) és a megszakításkezelő rutin címének betöltése az utasításszámlálóba következik. Utóbbi a processzor a memória utolsó háromszor két byte-jának valamelyikéből veszi a következő táblázatnak megfelelően:

\$FFFA - \$FFFB	Nem maszkolható megszakítás
\$FFFC - \$FFFD	Reset
\$FFFE - \$FFFF	Maszkolható megszakítás

A szubrutinok előbb-utóbb egy RTI utasítással fejeződnek be, amely visszaállítja a megszakítást megelőző állapotot, azaz visszamenti a veremből a P és az utasításszámláló értékét, és folytatja a program futását. A maszkolható megszakítást kezelő programok (és más megszakításra kényes tevékenységek) tartalmazzák a SEI (Set Interrupt Flag) a megszakítás maszkolására és a CLI (Clear Interrupt Flag) utasítást az újraengedélyezésre.

A megszakításokat általában hardware okok idézik elő, pl. az órajel, amely a C64-en minden 60-ad másodpercben megszakítást okoz (bár ez programból kikapcsolható).

De lehet megszakítást keltetni software uton, azaz program által is, a BRK (Break) utasítás végrehajtásával. Ez egy szabályos maszkolható megszakítást okoz, az egyetlen különbség a hardware okokból bekövetkezetthez képest, hogy az ST regiszter B jelzője magasra állítódik. Innen tudható a megszakítást kezelő szubrutinban, hogy a megszakítást BRK utasítás idézte elő. Ez az utasítás nagyon

hasznos a program hibajavítása során, mivel a hibás ágra futó programot a BRK jól követhetően megállítja. Ennek hasznosítására természetesen gondoskodni kell az \$FFFE által hivatkozott megszakításkezelő rutin megfelelő átírásáról. A gép bekapcsolást követő állapotban a BRK BASIC melegindítást hajt végre, hatása megegyezik a RUN/STOP és a RESTORE billentyűk egyidejű lenvomásával.



### 3. Assembly = kényelmes gépkód

#### 3.1 Mit tud egy assembler - mit kellene tudni egy assemblernek

Ha gépi kodu programozásról beszélünk, szinte mindig assembly programozásra gondolunk. Természetesen megvan a mód és lehetőség arra, hogy a gépkódú programot byte-ról byte-ra pl. a POKE BASIC utasítás segítségével elhelyezzük a memóriában és elindítsuk. Kétségtelenül ez a legfáradtságosabb és a legtöbb hibalehetőséget magában foglaló megoldás. Nem sok ennél kevésbé szórakoztató módja van a számítógépes időtöltésnek, éppen ezért ez a legrosszabb lehetőség, ha a program hossza az 1-2 utasítást meghaladja.

Szerencsére vannak a gépkódú programozásnak olyan eszközei, amelyek kombinálják a gépkód maximális hatékonyságát a gyors, biztonságos és jól nyomonkövethető munkával és ezek a különféle assemblerek. Az elképzeléseink szerint készült program azonban a Murphynek tulajdonított aforizma szerint nem vágyainknak, hanem utasításainknak megfelelően működik, ezért a megírt program még alapos csiszolásra szorul. Ebben nyújtanak segítséget a gépkódú monitorok, debuggerek, amelyekről szintén ejtünk pár szót a későbbiekben.

Előbb azonban az **assemblerokról**. Ezek tulajdonképpen olyan software-eszközök, speciális célprogramok, amelyek a gépkódú programok írására, fordítására, dokumentálására valók és alapvető céljuknak megfelelően a gépkódú programozás legegyszerűbb és egyszersmind leginkább hibamentes módját teszik lehetővé. Sokféle assembler van, amelyek különböző típusba sorolhatók, de egy-egy típuson belül is eltérnek szolgáltatásaikban és az elfogadott nyelv szintaxisát illetően.

Az assemblerek három fő típusa

- makróassemblerek,
- szimbólikus assemblerek és az
- azonnali (instant) assemblerek.

Szolgáltatásaik a fenti sorrendben egyre csökkennek. Az **azonnali assemblerek** (jobb gépkódú monitorok tartozéka) egyidejűleg egyetlen gépkódú utasítás fordítását végzik, szerepük az utasításmnemonikok és a címzésmódok felismerésére és lefordítására korlátozódik. A jobb megoldások a relatív címzésmódnál az ugráshoz szükséges gépkódú operandus kiszámítását is elvégzik, ha operandusként abszolút címet adunk meg.

A szimbólikus assemblerek már bonyolultabbak. Nem egy utasítást, hanem egy egész programot (vagy programrészt) egyszerre vizsgálnak meg. Az értékekre (cimek, operandusok) szimbólikus megnevezéssel hivatkozhatunk, ami nagymértékben leegyszerűsíti a programok javítását. Általában (több-kevesebb) lehetőséget adnak az adatterületek kijelölésére és kezdeti értékekkel való feltöltésére. Ezt az ún. direktívák segítségével teszik, amelyek az utasításkódhoz hasonló mnemonikok, de nem kerülnek be a lefordított programba, csak a fordítást vezérlő, az assemblernek szóló utasítások. Mivel az egyes utasítások lefordításához sokszor olyan információ is szükségeltetik, ami a program szövegében csak később következik, az assembler kénytelen többször is végigolvasni a programot, így a szimbólikus assembler fordítók majdnem mindig több (legalább két-) meneteseek.

A makróassembler a szimbólikus assembler funkcióin kívül a makródefiniálás és fordítás képességeivel is rendelkezik. A makrók fordítási időben működő programok, amelyet az assembler interpretál (mint a BASIC interpreter a BASIC kódot) és futásuk eredményeként assembly utasítások keletkeznek, amelyeket az assembler lefordít. Ez a kicsit talán bonyolult kép leegyszerűsítve úgy is jellemezhető, hogy a makrók kérdőívek, amelyeket a makróparaméterekkel kitöltve helyes assembly utasításokat nyerünk. Ennek a könyvnek nem célja a makróprogramozás oktatása, bár a témát még érintjük egy makróassembler ismertetése kapcsán.

A következőkben részletesen egy szimbólikus assembler példáján ismerjük meg az assemblerek és az assembly programozás lehetőségeit. Az egyértelmű tárgyalásmód kedvéért választanunk kellett egyet a C64-en rendelkezésre álló assemblerek közül, mivel, mint már említettük, azok szintaxisa és szolgáltatásai kismértékben eltérnek. A választás a talán leginkább elterjedt, mindenki keze ügyében lévő **HELP+BASIC bővítés** assemblerére esett, amely az alsó középmezőnyből származó, de ezért még használható programtermék. Kezelésére a 4.1 fejezetben visszatérünk, itt csupán azt kívánjuk leszögezni, hogy az ebben a fejezetben szerepeltetett szintaxis (meg a könyvben található összes programlista) a **HELP+ assembler**ére és csak arra vonatkozik, a többi assemblerre csak több-kevesebb módosítással igaz.

### 3.2 Utasításkészlet

Az assembly utasítások a következő formájúak:

<sorszám> <cimke> <mnemonik> <operandus> <megjegyzés>

A fenti elemek közül minden esetben kötelező a <sorszám> és a <mnemonik>, mindig elhagyható a <megjegyzés>, a mnemoniktól függően kötelező a <cimke> és az <operandus>. Egy sor csak sorszámból és megjegyzésből is állhat.

A <sorszám> csak a programok sorainak sorbarendezésére szolgál, tehát a programszerkesztés támogatására, az assembly nyelv szempontjából közömbös.

A <cimke> az utasítás vagy a direktiva azonosítására szolgál, az utasítás memóriacímét ill. a direktiva által kijelölt memóriacímét vagy értéket helyettesíti, ahelyett használható. Eppen erről kapta nevét a szimbolikus assembler.

A <mnemonik> egy gépikódu utasítás hárombetűs emlékeztető rövidítése vagy egy "-"tal bevezetett direktiva. Valamennyi utasítással és direktívával megismerkedünk a továbbiakban.

Az <operandus> értelmezése a mnemoniktól függ, általában címet vagy értéket jelöl. Az operandusból következtet az assembler az utasítások címzésmodjára is, a következőképpen:

<cimkifejezés>	- Utasításkódtól függően relatív vagy abszolút, utóbbi esetén, ha a cimkifejezés értéke <\$100, nullás lapu abszolút címzés.
(<cimkifejezés>)	- Indirekt.
(<cimkifejezés>),Y	- Indirekt indexelt.
(<cimkifejezés>),X	- Indexelt indirekt.
<cimkifejezés>,X és <cimkifejezés>,Y	- Abszolút indexelt.
#<cimkifejezés>	- Közvetlen.

Ha nincs operandus, beleértett címzésmodot használ az assembler (ha az utasításkód is megengedi). A ROL, ASL, LSR és ROR (ld. később) léptető utasítások beleértett modu címzése esetén (amikor is ezek az utasítások az A regiszterre vonatkoznak, az "A" operandust ki kell írni, ami egyébként cimkeként tilos az egyértelműség miatt. Az operandusban szereplő

<cimkifejezés> szimbólumok (címkek), decimális, hexadecimális (\$-ral kezdődő), bináris (%-kal kezdődő), oktális (&-cal kezdődő) szám, illetve ezek + és - jellel összekapcsolt kifejezése lehet. Speciálisan a <cimke> után (egyes assemblerek esetén elé) irt "<" jel a cím alacsonyabb, ">" jel a cím magasabb helyiértékű byte-ját jelenti. A fentiekre példát a későbbiekben még bőségesen találunk.

A <megjegyzés> ; -vel kezdődő tetszőleges karaktersorozat, amelyet az assembler csak kilistáz, egyébként figyelmen kívül hagy.

Most ismerkedjünk meg végre az utasításokkal is.

### 3.2.1. Töltőutasítások

LDA - a <cimkifejezés> által megjelölt byte tartalmát az akkumulátorba tölti  
Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: Z,N  
Példa:

LDA CIM

LDX - a <cimkifejezés> által megjelölt byte tartalmát az X regiszterbe tölti  
Alkalmazható címzés módok: KV,N,NY,A,AY  
Befolyásolt jelzők: Z,N  
Példa:

LDX CIM

LDY - a <cimkifejezés> által megjelölt byte tartalmát az Y regiszterbe tölti  
Alkalmazható címzés módok: KV,N,NX,A,AX  
Befolyásolt jelzők: Z,N  
Példa:

LDY CIM

### 3.2.2. Tárolóutasítások

STA - az akkumulátor tartalmát a <cimkifejezés> által megjelölt byte-ba menti  
Alkalmazható címzés módok: N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: -  
Példa:

STA CIM

STX - az X regiszter tartalmát a <cimkifejezés> által megjelölt byte-ba menti  
Alkalmazható címzés módok: N,NY,A  
Befolyásolt jelzők: -  
Példa:

STX CIM

STY - az Y regiszter tartalmát a <cimkifejezés> által megjelölt byte-ba menti  
Alkalmazható címzés módok: N,NX,A  
Befolyásolt jelzők: -  
Példa:

STY CIM

INC - a <cimkifejezés> által megjelölt byte tartalmát eggyel megnöveli  
Alkalmazható címzés módok: N,NX,A,AX  
Befolyásolt jelzők: Z,N  
Példa:

INC CIM

DEC - a <cimkifejezés> által megjelölt byte tartalmát eggyel csökkenti  
Alkalmazható címzés módok: N,NX,A,AX  
Befolyásolt jelzők: Z,N  
Példa:

INC CIM

### 3.2.3. Regiszterutasítások

TAX - az A regiszter tartalmát az X regiszterbe tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z,N  
Példa:

TAX

TAY - az A regiszter tartalmát az Y regiszterbe tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z,N  
Példa:

TAY

TXA - az X regiszter tartalmát az A regiszterbe tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z,N  
Példa:

TXA

- TYA - az Y regiszter tartalmát az A regiszterbe tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z, N  
Példa:  
TYA
- TXS - az X regiszter tartalmát a veremmutatóba tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: -  
Példa:  
TXS
- TSX - a veremmutató tartalmát az X regiszterbe tölti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: -  
Példa:  
TSX
- SEC - az átvitel-jelzőt magasra állítja  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: C  
Példa:  
SEC
- SED - a decimális mód-jelzőt magasra állítja  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: D  
Példa:  
SED
- CLC - az átvitel-jelzőt alacsonyra állítja  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: C  
Példa:  
CLC
- CLD - a decimális mód-jelzőt alacsonyra állítja  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: D  
Példa:  
CLD
- CLV - a túlszordulás-jelzőt alacsonyra állítja  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: V  
Példa:  
CLV
- CLI - a megszakítás-jelzőt alacsonyra állítja, azaz engedélyezi a maszkolható megszakításokat  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: I  
Példa:  
CLI



SEI - a megszakítás-jelzőt magasra állítja, azaz letiltja a maszkolható megszakításokat  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: I  
Példa:

SEI

DEX - az X regiszter tartalmát eggyel csökkenti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z, N  
Példa:

DEX

DEY - az Y regiszter tartalmát eggyel csökkenti  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z, N  
Példa:

DEY

INX - az X regiszter tartalmát eggyel növeli  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z, N  
Példa:

INX

INY - az Y regiszter tartalmát eggyel növeli  
Alkalmazható címzés módok: BE  
Befolyásolt jelzők: Z, N  
Példa:

INY

#### 3.2.4. Vezérlésátadó utasítások

JMP - az utasításszámláló értékét a <cimkifejezés> által meghatározott címre állítja, ez lesz a következő végrehajtandó utasítás  
Alkalmazható címzés módok: A, I  
Befolyásolt jelzők: -  
Példa:

JMP (\$304)

JSR - szubrutinhívás. Az utasításszámláló értékét kettővel megnöveli és ezt a címet a verembe menti. Ezután a programszámlálót a <cimkifejezés> által meghatározott címre állítja, ez lesz a következő végrehajtandó utasítás. Megjegyzendő, hogy a veremben nem a JSR-t követő utasítás címe van, hanem annál eggyel kisebb érték, de ezzel a programozónak általában nem kell törődnie, mivel ezt a szubrutinból való visszatérési utasítás (RTS) megfelelően lekezeli.

Alkalmazható címzés módok: A

Befolyásolt jelzők: -

Példa:

JSR SETTIM

RTS - szubrutinból való visszatérés. A verem két felső byte-jának eggyel megnövelt értékét az utasítás-számlálóba tölti. Ezzel a soronkövetkező végrehajtandó utasítás a JSR-t követő utasítás lesz a szubrutint hívó programrészben.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: -

Példa:

RTS

RTI - megszakításkezelő szubrutinból való visszatérés. Az állapotregisztert a verem tetején talált értékkel tölti. A verem soronkövetkező két byte-jának eggyel megnövelt értékét az utasításszámlálóba tölti. Ezzel a soronkövetkező végrehajtandó utasítás a megszakítást követő utasítás lesz.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: valamennyi

Példa:

RTI

BRK - megszakítást okozó utasítás (software megszakítás). A BRK utasítást követő byte címe a verem tetejére kerül, majd az állapotregiszter értéke is kimentődik. Ezután a vezérlés a \$FFFE-\$FFFF cím által mutatott utasításra kerül. Letiltja a maszkolható megszakításokat.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: B, I

Példa:

BRK

BCC - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha az átvitel-jelző alacsony.

Alkalmazható címzés módok: RE

Befolyásolt jelzők: -

Példa:

BCC CIKLUS

BCS - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha az átvitel-jelző magas.

Alkalmazható címzés módok: RE

Befolyásolt jelzők: -

Példa:

BCS CIKLUS

- BEQ - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a nulla jelző magas.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BEQ CIKLUS
- BNE - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a nulla jelző alacsony.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BNE CIKLUS
- BMI - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a negatív jelző magas.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BMI CIKLUS
- BPL - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a negatív jelző alacsony.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BPL CIKLUS
- BVC - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a túlcsordulás jelző alacsony.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BVC CIKLUS
- BVS - az utasításszámláló tartalmához előjelhelyesen hozzáadódik az operandus tartalma, ha a túlcsordulás jelző magas.  
Alkalmazható címzés módok: RE  
Befolyásolt jelzők: -  
Példa:  
BVS CIKLUS

### 3.2.5. Veremkezelő utasítások

FHA - az akkumulátor tartalmát a verembe menti, a veremmutató által mutatott címre, majd a veremmutató értékét eggyel csökkenti.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: -

Példa:

FHA

PLA - a veremmutató által hivatkozott címen lévő értéket az akkumulátorba tölti, majd a veremmutató értékét eggyel megnöveli.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: -

Példa:

PLA

PHP - az állapotregiszter tartalmát a verembe menti, a veremmutató által mutatott címre, majd a veremmutató értékét eggyel csökkenti.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: -

Példa:

PHP

PLP - a veremmutató által hivatkozott címen lévő értéket az állapotregiszterbe tölti, majd a veremmutató értékét eggyel megnöveli.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: valamennyi

Példa:

PLP

### 3.2.6. Leptető utasítások

ASL - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozícióval balra lépteti. A legalsó bit nulla lesz, a kilépő bit az átvitel-jelzőbe kerül. Az utasítás 2-vel való szorzásnak felel meg. A ROL utasítással kombinálva kétbyte-os érték szorzását lehet elvégezni.

Alkalmazható címzés módok: BE, N, NX, A, AX

Befolyásolt jelzők: Z, N, C

Példa:

ASL A

vagy

ASL HATVNY

ROL - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozícióval balra lépteti. A legalsó bitpozícióra az átvitel-jelző utasítás előtti értéke lép, a kilépő bit az átvitel-jelzőbe kerül. Az utasítás 2-vel való szorzásnak felel meg. Az ASL utasítással kombinálva kétbyte-os érték szorzását lehet elvégezni.

Alkalmazható címzés módok: BE, N, NX, A, AX

Befolyásolt jelzők: Z, N, C

Példa:

ROL A  
vagy  
ROL HATVNY

LSR - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozícióval jobbra lépteti. A legfelső bit nulla lesz, a kilépő bit az átvitel-jelzőbe kerül. Az utasítás 2-vel való osztásnak felel meg. A ROR utasítással kombinálva kétbyte-os érték osztását lehet elvégezni.

Alkalmazható címzés módok: BE, N, NX, A, AX

Befolyásolt jelzők: Z, N, C

Példa:

ASR A  
vagy  
ASR HATVNY

ROR - az akkumulátor vagy a memóriabyte tartalmát egy bit-pozícióval jobbra lépteti. A legfelső bitpozícióra az átvitel-jelző utasítás előtti értéke lép, a kilépő bit az átvitel-jelzőbe kerül. Az utasítás 2-vel való osztásnak felel meg. Az ASR utasítással kombinálva kétbyte-os érték osztását lehet elvégezni.

Alkalmazható címzés módok: BE, N, NX, A, AX

Befolyásolt jelzők: Z, N, C

Példa:

ROR A  
vagy  
ROR HATVNY

### 3.2.7. Logikai utasítások

AND - az akkumulátor és a specifikált adatbyte között logikai ES művelet végzése bitenként. Az eredmény az akkumulátorba kerül. Az eredmény valamely bitje magas, ha mindkét megfelelő bit magas volt, alacsony minden egyéb esetben. Ezáltal egy memóriabyte bizonyos bitjeit lehet az akkumulátorba vinni, míg a többi bit 0 lesz.

Ezt a byte maszkolásának is szokás nevezni.  
Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: Z,N  
Példa:

```
LDA #$7F
AND BYTE
```

ORA - az akkumulátor és a specifikált adatbyte között logikai VAGY művelet végzése bitenként. Az eredmény az akkumulátorba kerül. Az eredmény egy bitje magas, ha legalább az egyik bit magas volt, alacsony ha mindkettő alacsony volt. Ezáltal két byte tartalmát lehet egyesíteni az akkumulátorban.

Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: Z,N  
Példa:

```
LDA #$0F
AND BYTE1
STA BYTE1
LDA #$F0
AND BYTE2
ORA BYTE1
```

EOR - az akkumulátor és a specifikált adatbyte között logikai kizáró VAGY művelet végzése bitenként. Az eredmény az akkumulátorba kerül. Az eredmény egy bitje magas, ha pontosan az egyik bit magas volt, alacsony minden egyéb esetben. Az utasítás invertálja az egyik byte azon bitjeit, ahol a másik byte-ban 1 áll.

Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: Z,N  
Példa:

```
LDA #$FF
EOR BYTE1 ; LOGIKAI KOMPLEMENTIS-KEPZES
```

BIT - az akkumulátor és a specifikált adatbyte között logikai ES művelet végzése bitenként. Az akkumulátor értéke nem változik a művelet végrehajtása során. A nulla jelző magas lesz, ha az eredmény nulla, a negatív-jelző a memóriabyte 7., a túlcsordulás-jelző a 6. bit értékét fogja kapni. Az utasítással előnyösen lehet a 6. és 7. bit értékét vizsgálni.

Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY  
Befolyásolt jelzők: Z,N,V  
Példa:

```
LDA #$FF
BIT BYTE
BMI BIT7
BVS BIT6
BEQ URES
```



### 3.2.8. Aritmetikai utasítások

ADC - összeadó utasítás. Az akkumulátor, a megcímzett adatbyte és az átvitel jelző értékét összeadja és az akkumulátorba viszi. Ha a decimális-jelző magas, a számolást BCD alakban végzi (ritkán használatos a C64 esetében). Az átvitel-jelző hozzáadásának a többbyte-os összeadás megvalósításában van szerepe, ha egyetlen vagy az első byte értékét kívánjuk kiszámolni, az utasítás előtt az átvitel jelzőt törölni kell.

Alkalmazható címzés módok: KV, N, NX, A, AX, AY, IX, IY

Befolyásolt jelzők: Z, N, V, C

Példa:

```
CLC
ADC MEM
```

SBC - kivonás. Az akkumulátorból kivonja a megcímzett adatbyte-ot és az átvitel jelző értékének inverzét és az eredményt az akkumulátorba viszi. Ha a decimális-jelző magas, a számolást BCD alakban végzi (ritkán használatos a C64 esetében). Az átvitel-jelző levonásának a többbyte-os kivonás megvalósításában van szerepe, ha egyetlen vagy az első byte értékét kívánjuk kiszámolni, az utasítás előtt az átvitel jelzőt magasra kell állítani.

Alkalmazható címzés módok: KV, N, NX, A, AX, AY, IX, IY

Befolyásolt jelzők: Z, N, V, C

Példa:

```
SEC
SBC MEM
```

### 3.2.9. Összehasonlító utasítások

CMF - összehasonlítja az akkumulátor és a specifikált adatbyte tartalmát, oly módon, hogy az utóbbit kivonja az előbbiből, de az akkumulátor tartalma nem változik, viszont a jelzőket az eredménynek megfelelően állítja. Ez azt jelenti, hogy az összehasonlítás eredményét az alábbi utasításokkal lehet vizsgálni:

```
BPL -> A >= memoriabyte
BMI -> A < memoriabyte
BEQ -> A = memoriabyte
BNE -> A <> memoriabyte
```

Alkalmazható címzés módok: KV,N,NX,A,AX,AY,IX,IY

Befolyásolt jelzők: Z,N

Példa:

```
CMP #$FO
BNE CIM1
```

- CPX - Összehasonlítja az X regiszter és a specifikált adatbyte tartalmát, olyan módon, hogy utóbbit kivonja előbbiből, de az akkumulátor tartalma nem változik, viszont a jelzőket az eredménynek megfelelően állítja.

Alkalmazható címzés módok: KV,N,A

Befolyásolt jelzők: Z,N

Példa:

```
CPX #$FO
BNE CIM1
```

- CPY - Összehasonlítja az Y regiszter és a specifikált adatbyte tartalmát, olyan módon, hogy utóbbit kivonja előbbiből, de az akkumulátor tartalma nem változik, viszont a jelzőket az eredménynek megfelelően állítja.

Alkalmazható címzés módok: KV,N,A

Befolyásolt jelzők: Z,N

Példa:

```
CPY #$FO
BNE CIM1
```

- NOP - üres utasítás, teljesen hatástalan.

Alkalmazható címzés módok: BE

Befolyásolt jelzők: -

Példa:

```
NOP
```

#### 4. Software eszközök - assemblerek, monitorok, debuggerek

Az assembly programok írásához szerkesztőprogramokat, lefordításukhoz assemblereket használunk. Programszerkesztésre egyes assemblerek esetén a BASIC képernyőszerkesztőjét alkalmazhatjuk (mintha csak BASIC programot írnánk), más esetekben viszont speciális szerkesztőprogramokat. Az assembler is lehet külön program, vagy egy általánosabb fejlesztő rendszer egyik funkciója, mindkettőre láthatunk példát a későbbiekben.

A programozás természetesen nem ér véget a program lefordításával, bizonyos esetekben ez csak az első lépés. Ezután következik a program kipróbálása, tesztelése, javítása egészen addig, míg meg nem győződünk arról, hogy valóban azt a feladatot hajtja végre, amelyre terveztük. Ennek a munkának a során is legalább olyan mértékben rá vagyunk szorulva software-eszközök segítségére, mint a program írása és fordítása alatt. Ugyanakkor ennek a tevékenységnek a támogatására igen nehéz kidolgozni általános célú eszközöket, mivel a tesztelendő programok sokkal változatosabb feladatok megoldására lettek tervezve, mint egy átlagos BASIC program. Gondoljuk csak meg, milyen súlyos problémákat vet fel egy gépkódu monitor tesztelése, hiszen nagyon nehéz úgy elemezni működését, hogy ne avatkozzunk abba bele. Ennek köszönhetően, különösen a személyi számítógépek területén csak általános és elég egyszerű eszközök terjedtek el. Ezek a **monitorok** és az ezek szolgáltatásain némileg túlmenő **debuggerek** (jobb magyar szó híján). Mindkettőre látunk példát ebben a fejezetben. Látszólagos egyszerűségük ellenére is igen jó szolgálatot tesznek egy-egy új program készítésénél. Nagyobb lélegzetű munkák esetén azonban igen lényegessé válik az egyedi, feladatra szabott teszt-ág és a részletes tesztelési terv elkészítése. Ez nem mindig fedhető le a debuggerek szolgáltatásaival és bizonyos feladatok esetén magának a problémának a megoldásával azonos mértékű programozási feladatot jelent. Elhagyásával viszont igen rossz minőségű, megbízhatatlan programterméket nyerünk, amellyel több utólagos hibajavítási feladatunk lesz, mint amennyi munkát a tesztelés következetes megoldására kellett volna szánni.

#### 4.1 A HELP+ assembly programozást segítő eszközei

A HELP+ program elég jól ismert a hazai C64 felhasználók körében. Bár valamennyi területen szolgáltatásai inkább csak átlagosak, nagy előnye, hogy a software-fejlesztés elég széles körét, eszköztárát átfogja. Itt csak az assembly programozáshoz kifejezetten hozzátartozó szolgáltatásait tárgyaljuk, hiszen ez könyvünk témája.

A HELP+ betöltése és elindítása után a BASIC szerkesztőjébe lépünk be, amely lehetővé teszi, hogy a HELP+ assembléere számára forrásprogramot írjunk. Ennek menete teljesen megegyezik a BASIC programok rögzítésével, persze itt szintaktikusan helyes assembly utasításokat kell írunk. Ha forrásprogramunk elkészült, azt mentjük ki a BASIC programnál megszokott módon egy lemezfile-ba.

Az **assembly** utasítások írásmódjáról már szoltunk, most ismerjük meg a HELP+ assemblerének direktíváit is. A direktívák az utasításokkal ellentétben az assemblernek szóló parancsok és a fordítás folyamatát vezérlik. Általában a műveleti kód mezőben szerepelnek és pont karakter vezeti be őket.

**.BYTE** - egy byte tárolóterületet lefoglal és azon elhelyezi a direktíva után álló kifejezés értékét. Ha vesszővel elválasztva több értéket adunk meg, a megfelelő számú byte kerül lefoglalásra. A kifejezés megengedett értéke 0 és 255 között van, a határokat is beleértve.

**.WORD** - mint a **.BYTE**, de egy szót, két egymást követő byte-ot foglal le a C64-nél megszokott alacsony helyiértékű byte - magas helyiértékű byte sorrendben.

Igy pl.

```
CIM .WORD $B7F7
```

\$F7, \$B7 sorrendben lesz lerakva.

**.TEXT** - a direktívát aposztrófok vagy idézőjelek között követő string karaktereinek ASCII kódjai lesznek lerakva az aktuális címtől kezdve.

**.DISP** - mint **.TEXT**, de 6 bites ASCII kódok kerülnek lerakásra (azaz C64 screen-kódok).

**.END** - a forrásprogramot záró direktíva, ha nem tesszük ki, az assembler a forráskód után generálja 65535-ös látszólagos sorszámmal.

**.LOAD** - a direktíva után irt nevű programmal folytatódik a fordítás folyamata. Az assembler "kedves" hibája folytán a név hosszának pontosan meg kell egyeznie az éppen fordítás alatt álló programéval.

= - (egyenlőségjel) a szimbólum értékadás direktívája. Az előtte (a címke mezőben) álló szimbólum az utána (az operandus mezőben) álló kifejezés értékét veszi fel. Mind előtte, mind utána szerepelhet a "\*" szimbólum, amely a fordítás aktuális címét tartalmazza, azaz azt a memóriacímét, ahova a soronkövetkező byte-ot el fogja helyezni az assembler. Néhány példa:

```
*=$C000
```

A következő utasítás vagy adat a \$C000-ra kerül.

```
ALFA=*
```

ALFA az elhelyezés-mutató aktuális értékét veszi fel.

```
TEXT .TXT 'ABCDE'  
THOSSZ=*-TEXT
```

THOSSZ a szöveg hosszának értéket kapja.

```
*=**+20
```

Ez is szabályos. 20 byte lefoglalását jelenti.

**Az assembler elindítása** a sor elejére írt "[" karakterrel kezdődik, ezután az assembler három kérdést tesz fel:

- az első kérdésre a lemezfile nevével kell válaszolni, amely a forrásprogramot tartalmazza,
- a második kérdés ("Hexa korr. poke:") lehetőséget ad arra, hogy a lefordított gépiódu program ne a saját helyére, hanem az itt megadott értékkel kisebb (!) címre kerüljön. Erre pl. akkor van szükség, ha a program tervezett helye a HELP+ által lefoglalt \$8000-tól kezdődő 16 Kb-ba esik. Ha itt "RETURN"-t adunk, a program a forrásban előírt helyre kerül,
- a harmadik kérdésre egy háromjegyű bináris számmal kell válaszolni, amely a listázásra utal. Az első jegy helyén álló 1-essel hibajelzéseket, a második pozíción állóval forráslistát, a harmadikkal a szimbólumtábla kinyomtatását válthatjuk ki. Utobbi a címkék neve és a hozzájuk rendelt érték hexadecimálisan. Bekapcsolt nyomtató esetén a fordítási lista ott is megjelenik.

Az assembler felhasználja a BASIC területet (\$801-től) a szimbólumtábla tárolásához, így a memóriában lévő program törlődik.

Sajnos a lefordított program lemezre mentéséhez vajmi kevés segítséget nyújt a HELP+. A fordítás eredménye a tárba kerül, a forrásprogramban meghatározott helyre. A fordítási listából (és csak abból) meghatározható a program első és utolsó utasításának a címe, azaz a program kezdete és vége. Ha a memória \$2B-\$2C címére beírjuk a program első byte-jának a címét, míg a \$2D-\$2E címére az utolsó byte-énál

eggyel nagyobb címet (a .END direktiva mellé irt cím a listában) a BASIC programterület kezdetét és végét jelző ponttereket állítottuk át. Az ezt követő SAVE BASIC utasítás éppen lefordított programunkat viszi lemezre. A pontterek átállítását akár POKE BASIC utasítással, akár a HELP+ monitor funkciójával végezhetjük. Utóbbival egyszerűbb, mivel ebben az esetben a hexadecimális értéket nem kell decimálisra átszámolni. A program kimentése után ne felejtsük el a \$2B-\$2C és \$2D-\$2E mutatókat egyaránt \$801-re, a BASIC terület elejére állítani, majd egy NEW-t végrehajtani, mert egyébként nem kívánt módon átirhatjuk a memóriát. Ehelyett egy SYS 64738 utasítással szintén alaphelyzetbe hozhatjuk a BASIC interpretert.

A következőkben foglalkozzunk az általunk ismertetésre kerülő programtesztelési eszközök legegyszerűbbikével, a HELP+ monitor funkciójával. Ezek sajnos igen szegényesek, ezért rájuk is igaz, ami a HELP+ assemblerére, csak igen kisméretű programok készítésénél nyújtanak értékelhető segítséget.

A monitorba a "]" karakter sor elején való kiadásával léphetünk be. Ekkor rögtön megjelenik a képernyőn a legutoljára kezelt memóriabyte címe és értéke hexadecimális szám formájában. Ezen aktuális cím értékét a "+" karakter után beírt (mindig hexadecimálisán értendő) értékkel változtathatjuk meg. A "RETURN" megnyomásával a következő byte-ra ugrik a monitor. A memória tartalmát a "/" megnyomása után írt értékkel megváltoztathatjuk. A "RETURN"-nel a következő byte-ra áttérve módosítási módban maradunk egészen egy újabb (pl. "+") parancs kiadásáig. Legyünk nagyon óvatosak a memóriamódosítás kipróbálása során! Az első két lap bizonyos címeinek és persze a HELP+ által elfoglalt terület (\$8000-\$A000) megváltoztatása kellemetlen következményekkel járhat. A monitoron belül a "[" kiadásával a nyomtatóra, a "]" megnyomásával a képernyőre irányíthatjuk az outputot. Hasonlóan kapcsoló szerepet játszanak a "<" és ">" jelek. "<" kiadásával a géphez csatlakoztatott VC 1541-es lemezegységének memóriájára kapcsolunk át, ezután minden művelet arra és nem a C64 központi tárára vonatkozik egészen a ">" utasítás kiadásáig, ami helyreállítja az eredeti állapotot. A "\*" karakterrel az aktuális címen kezdődő gépikódu programra adódik a vezérlés. Megjegyzendő, hogy ez a parancs lemezegység-módban (tehát "<" után) is működik, de akkor a lemezegység processzorára és tárolójára vonatkozik. Kiadásával természetesen óvatosan kell bánni.

A monitor maradék funkciói a disassemblerre vonatkoznak. Ennek segítségével a memória tartalmát assembly formára emlékeztető listává lehet visszairni. A visszairhatóság természetesen korlátozott, nem vonatkozhat a címkére és megjegyzésre, hiszen ezeknek semmi nyomuk a gépikódu programban. A "-" paranccsal az aktuális címtől kezdődő egyetlen utasítást disassemblálhatunk. A szóköz billentyű megnyomásával az aktuális címtől kezdődő folyamatos visszafordítást indíthatunk meg, amely egészen a "RUN/STOP" megnyomásáig tart. Hosszabb programrész visszafordítását természetesen

tesen célszerű nyomtatóra végezni, így utólag kényelmesebben elemezhetjük az eredményt. A C64 és a VC 1541 ROM-jának tanulmányozásához azonban csak végső esetben ajánljuk ezt a módszert, mivel ebben a vonatkozásban igen jó könyvek állnak rendelkezésre, ha nem is magyar nyelven (ld. függelékek).

A HELP+ monitorából a BASIC-hez az "=" paranccsal térhetünk vissza.



## 4.2 Makróassembler fejlesztő rendszer C64-re

A Commodore Business Machines által forgalmazott fejlesztő rendszer leírását inkább szemléltetőnek, eligazításnak szánjuk, semmint kézikönyvnek és különösen nem tankönyvnek, mivel a teljes leírás által megkívánt terjedelem messze meghaladja azt, amit e könyv keretében részánhatunk. Igaz ez a 4. fejezet soronkövetkező alpontjaira is.

A makróassembler fejlesztő rendszer már sokkal komolyabb programfejlesztő eszköz az előzőekben megismert HELP+-nál. Az assembler fordítóprogram makrók használatát is megengedi, gazdagabb hibajelzéssel rendelkezik, használata kényelmesebb. A DOS-szal, a forrásnyelvi szerkesztővel, a keresztreferencia készítővel, a két töltőprogrammal és a monitorokkal együtt kerek rendszert képez, jól használható nagyobb lélegzetű assembler programok készítése során is. Bár – mint a későbbiekben látni fogjuk – léteznek jobb, az assembly programozást jobban támogató eszközök is, igen elterjedt a C64 programozók körében.

A programírás és fordítás eszközeit a "BOOT ALL" programmal tolthetjük be. Ezután az "EDITOR64" program vezérlése alá kerülünk. A látszat az, mintha szokványos BASIC rendszerben lennénk, de ez nem igaz, mert az EDITOR64 alatt a programsorok interpreter-kódba váltása nem történik meg, ezért az ekkor beírt BASIC programok hibásak lesznek. Az EDITOR64 tehát csak assembly nyelvű programok bevitelére és javítására alkalmas. A program szerkesztésében használhatjuk a BASIC-képernyőszerkesztő minden lehetőségét és még néhány igen praktikus kiegészítést. Ezek dióhéjban a következők:

- AUTO     -    automatikus sorszámprompt.  
Az AUTO n mindaddig sorszámpromptot ad az új sor elejére n növekménnyel, míg az AUTO (operandus nélkül) parancsot ki nem adjuk.
- NUMBER -    sorok újraszámozása.  
A NUMBER n1, n2, n3 parancs újraszámozza n1-től n2 kezdőértékkel n3 növekménnyel a program sorait.
- GET       -    töltés lemezről.  
A GET "név", n1, n2, n3 parancs a "név" nevű lemezes file tartalmával az n1 sorszámú, már a memóriában lévő sortól kezdve felülírja a programok tárolására szolgáló területet. n2 és n3 az egység- és csatornaszám, alapértelmezés szerint 8 (ha nem adjuk meg). Az n1 is elhagyható, de ha megadjuk, a parancs felhasználható forrásfile-ok egymás után fűzésére is.

- PUT - mentés lemezre.  
A PUT "név", n1-n4, n2, n3 parancsban n2 és n3 jelentése, mint GET-nél. Ha n1-et és esetleg n2-t is megadjuk, a parancs csak a két sorszám közé eső programrészt menti a "név" nevű file-ba.
- CPUT - tömörített mentés.  
paraméterei és használata megegyezik a PUT-éval, de a file-t tömörítve, az összes felesleges space elhagyásával menti.
- FIND - karaktersorozat keresése.  
A FIND /str/, n1-n2 utasítás az str karakterlánc valamennyi előfordulását kilistázza a teljes programban, vagy n1 és és/vagy megadása esetén a jelzett sortartományban. A "/" karakter helyett tetszőleges, a stringben nem szereplő delimiter használható.
- CHANGE - szövegcsere.  
A CHANGE /str1/str2/, n1-n2 úgy működik, mint a FIND, de a megadott sorszámtartományban (vagy a teljes programban) az str1 valamennyi előfordulását str2-re cseréli.
- DELETE - sorok törlése.  
A DELETE n1-n2 parancs (szintaxisa mint a LIST-é) a megadott sorszámtartomány sorait törli a memóriából.
- FORMAT - formázott lista.  
A FORMAT n1-n2 parancs a listázást formázva végzi, azaz külön oszlopban kezdi a címet, utasítás-mnemonikokat, operandusokat és megjegyzéseket az assembler listához hasonlóan. Kicsit buta, így címkének tekint minden, az első pozíción kezdődő szót és mnemoniknak minden sor első szavát, ha nem az első oszlopban kezdődik.
- KILL - kikapcsolja az EDITOR64-et, visszatér a BASIC-editorba. Az editor egyébként a SYS 49152 BASIC paranccsal indítható újra.

Az assembly program megírása során lényegében ugyanazt a formátumot kell alkalmazni, mint a HELP+ esetén az alábbi eltérésekkel. A kifejezésekben a + és - jelek mellett a \* és a /, azaz szorzás és osztás is használható. Az operandusmezőt követő megjegyzések előtt nem kötelező kitenni a pontosvesszőt, de ajánlatos az egységesség kedvéért, hiszen nem minden utasításnak van operandusa. A cím alsó ill. felső byte-jára a szimbólum elé irt < vagy > jellel utalhatunk. Az operandus nemcsak szám, ill. számtani kifejezés lehet, hanem

egy "'" (aposztróf) jellel megelőzött karakter is, ekkor az operandus értéke a karakter ASCII kódja. A direktívák köre szélesebb, mint a HELP+-nál. Az ugyanolyan jelentésű .BYTE, .WORD, .END és az "=" jel mellett a következők használhatók:

- .DBYTE - ugyanugy 2 byte elhelyezésére szolgál, mint a .WORD, de a byte-okat magas helyiértékű byte - alacsony helyiértékű byte sorrendben rakja le, eltérően a 6510-es konvenciótól.
- .PAGE - lapdobást okoz az assembler-listában. A direktíva után aposztrófok közé írt szöveg a direktívát követő minden lap tetejére kiíródik, amíg egy újabb szöveggel ellátott .PAGE nem jön.
- .SKIP - az utána álló kifejezés értékének megfelelő -ha hiányzik, egyetlen- sort emel az assembler listában. Sem a .SKIP, sem a .PAGE nem jelenik meg a listán.
- .OPT - a fordítási opciók beállítására szolgál a fordítás közben. A lehetséges opciók:
  - ERRORS - NOERRORS - hibafájl keletkezése
  - LIST - NOLIST - listázás
  - GENERATE- NOGENERATE - a stringek kinyomtatásaElegendő az opciók első 3 karakterét megadni.
- .FIL - hatásában megegyezik a HELP+ .LOAD direktívájával, azaz a fordítás a direktíva után álló nevű lemezes fájl tartalmával folytatódik.
- .LIB - úgy működik, mint a .FIL, de a könyvtári fájl végigolvasása után az assembler folytatja a félbehagyott fájl fordítását. A .LIB-bel aktivizált fájl nem tartalmazhat .LIB-et, de .FIL-t igen. Nagyon hatékony eszköz szubrutin-könyvtárak létrehozására!

A maradék direktívák a makródefinícióval kapcsolatosak. Ezeket egy kicsit részletesebben tárgyaljuk és közben megpróbálunk bepillantani a makróprogramozás rejtelmeibe, ami jól alkalmazva igencsak megnöveli programozói hatékonyságunkat.

**A makróprogramozás** tulajdonképpen makródefiníciókból és makróhívásokból áll. Ezek az elemek szabadon keverhetők az assembly sorokkal. A makró-definíció a következő szerkezetű:

```
<címke>      .MAC      <makrónév>
              <utasítások>
<címke>      .MND
```

A címkék elhagyhatók, egyébként a makró által generált első és utolsó utasításhoz tartoznak. Az <utasítások> tetszőleges szöveget tartalmazhatnak, kivéve a .MAC és .MND direktívákat. A makróhoz legfeljebb 9 db paraméter tartozhat. A formális paramétereket a makró-definícióban ?1, ?2 ... ?9-cel jelöljük. Ezek úgynevezett pozicionális paraméterek, azaz az ?1 utal a makróhívásban szereplő első, a ?2 a második stb. aktuális paraméterre. A paraméterek szövegtípusúak, azaz a formális paraméterek a makródefinióban a makróhívásban szereplő aktuális paraméterek szövegével helyettesítődnek, minden formai ellenőrzés nélkül. Ha az assembler egy

<makrónév> <paraméterek>

alakú makróhívást talál a programban, ahol a <paraméterek> szövegek vesszővel elválasztott sorozata, a következőt teszi:

- sorra veszi a makródefiniáció sorait,
- egy adott sorban minden formális paramétert szöveg-szerűen helyettesít a hívásban sorszámának megfelelő aktuális paraméter-stringgel. Ha nem adtunk meg egy, a makródefinióban hivatkozott paramétert, az assembler egy Lxxx alakú paraméter-stringet generál, ahol xxx 001-től folyamatosan növekvő szám. Ez biztosítja, hogy a generált paraméterek minden híváskor különbözőek egymástól.
- ha minden behelyettesítést elvégzett, az assembler úgy tekinti a sort, mintha az a forrásfile-ból jött volna, azaz lefordítja.
- a makródefiniáció utasításai között természetesen a behelyettesítések után előállhat makróhívás is, az ilyen hívások azonban legfeljebb 8-as mélységig ágyazhatók egymásba. A makró hívhatja önmagát is.

Ez persze így egy kicsit túl tömény, lássunk néhány példát. Ezek a példák tulajdonképpen az 5. fejezetbe illenének, amely a programozás technológiájával foglalkozik, tekintsük őket egy kis előrelapozásnak. Az assembly nyelvű programozás során különösen gyakran fordul elő, hogy ugyanazokat az utasításokat írjuk le, egymástól alig-alig különböző elemekkel. Az ilyen utasítás-sorozatok az assembly programozás elemi építőkövei, amelyek szinte makróba kívánnakoznak. Pl. két egybyte-os szám összeadása előtt törölnünk kell az átvitel-jelzőt (carry), ha helyes eredményt akarunk kapni. Eszerint a  $C=A+B$  összeadást

```
LDA A
CLC
ADC B
STA C
```

módon írhatjuk le.

Ha a feladat történetesen  $X=Z+22$ , az utasítássorozat:

```
LDA Z
CLC
ADC #22
STA X
```

A hasonlóság szembeszökő. Sokkal egyszerűbb, ha írunk egy makrót:

```
.MAC OSSZ1B
LDA ?1
CLC
ADC ?2
STA ?3
.MND
```

alakban, amelyet ha a következőképpen hívunk:

```
OSSZ1B A,B,C
```

éppen az első, ha így:

```
OSSZ1B Z,#22,X
```

a második példát kapjuk a makró kifejtésekor.

Egy másik, kicsit összetettebb illusztráció az igen gyakran szükséges ciklusszervezést könnyíti meg:

```
.MAC CFEJ
LDX ?2
?1K CPX ?3
BEQ ?4
BPL ?1V
?4 .MND
```

```
.MAC CVEG
INX
JMP ?1K
?1V .MND
```

A két makróból álló együttes egy alkalmazása:

```
LDA #0
CFEJ C1,#1,#10
STA ALFA,X
CVEG C1
```

## A makróprogram kifejtése:

```
          LDA      #0
          LDX      #1
C1K       CPX      #10
          BEQ      L001
          BPL      C1V
L001      STA      ALFA,X
          INX
          JMP      C1K
C1V
```

A CFEJ makrónak három paramétere van, a ciklus neve - ebből képezzük a ciklusfej és a ciklusvég címkeit -, a ciklusváltozó kezdőértéke és a maximális ciklusváltozó érték, amelyre még le kell futtatni a ciklust. A makrók természetesen feltételezik, hogy az X regisztert, amit ciklusváltozóként alkalmazunk, a ciklustörzs utasításai nem módosítják. Az alkalmazási példa egy memóriatömb nullázása (ALFA+1-től ALFA+10-ig). A CFEJ makró hívásakor hiányzó negyedik paraméter (?4) helyére az assembler egy címkét generált, mégpedig olyat, amely a CFEJ egy másik hívásában generálttól feltétlenül különbözik, ezzel biztosítja a címkek egyedi voltát. A ciklusvég természetesen ugyanazt az első paraméter kapja, így válik lehetővé a visszaugrás a ciklus elejére, a generált C1K címkeire. Megjegyzendő még, hogy a ciklus felső ciklusvég-vizsgálatot alkalmaz, ha tehát belépéskor a ciklusvég-feltétel nem elégül ki (a ciklusváltozó kezdőértéke nagyobb mint a végérték) a ciklus egyszer sem fut le.

Reméljük a fenti példák felkeltették az olvasó érdeklődését a makróprogramozás iránt. Mielőtt azonban továbbhaladnánk a makroassembler fejlesztői rendszer megismerésében, nem szabad elhallgatnunk azt, hogy az ismertetett makroassembler talán minden makroassemblerek legszegényesebb példája. A kicsit bonyolultabb feladatok megoldásához rettenetesen hiányoznak a makróváltozók (lokális és globális egyaránt). Ugyanúgy szegényes a makrókonyvtár kialakítási lehetőség is (egyedül a .LIB direktíva áll rendelkezésünkre), bár sok nagygépes makroassembler sem nyújt többet! Vegyük viszont figyelembe a programtermék megítélésénél, hogy más assembler készítők a makrólehetőséggel még csak nem is foglalkoznak.

Az editor segítségével megírt és lemezzre mentett assembly programot a következőkben leírt módon fordíthatjuk le.

Elsőször töltsük be az ASSEMBLER64 programot és RUN-nal indítsuk el. A fordítás során az assembler két file-t hoz létre, a betöltő programok számára készített tárgyprogram-file-t és a címkehivatkozásokat tartalmazó file-t, amelyből a CROSSREF64 programmal készíthetünk keresztreferencia-listát. Az assembler néhány kérdéssel indul, amelyek (és az

elvárt információ):

OBJECT FILE	- lefordított program file neve Ha a név üres, nem készül OBJECT file, csak ellenőrzés és/vagy keresztreferencia
HARD COPY	- sornyomtató lista kell vagy nem kell
CROSS REFERENCE	- keresztreferencia-file készül-jön-e. Megjegyzés: Ha a program tartalmaz .LIB direktívát, az OBJECT file és a keresztreferencia egyszerre nem készíthető el!

SOURCE FILE NAME - forrás file neve

Ha az assembler futását meg akarjuk szakítani, hogy a képernyőn megjelenő listát megvizsgáljuk, ezt a RUN/STOP billentyű megnyomásával érhetjük el. A továbbindítás bármilyen billentyűvel lehetséges, kivéve a B-t, amely befejezi az assembler futását és visszatér a BASIC operációs rendszerhez.

Az assembler által előállított tárgyprogramot a **két töltőprogram** valamelyikevel tölthetjük a memóriába (viqyázat, a tárgyprogram még nem végrehajtható program!).

A két program működésében azonos, de más helyre töltődik a memóriába ( \$800-ra ill. \$C800-ra), így megkönnyíti a felhasználói program tetszőleges helyre töltését.

Betöltés (LOAD "töltőnév",8,1) után a **LOLOAD64** RUN, a **HILOAD64** SYS 51200 utasítással indítható. A töltőprogramok elfogadnak egy eltolási cím (offset) értéket is, így lehetővé válik, hogy a programokat az eredeti cím + eltolási cím helyre töltsük be. A töltőprogramok lefutása után a felhasználói program futásra kész.

A töltőprogramokhoz hasonlóan a fejlesztői rendszernek **két monitorprogramja** is van, amelyek szintén csak a betöltési címekben különböznek egymástól, így lehetővé téve a memória tetszőleges helyén lévő (nem túl hosszú) program vizsgálatát. Az XVM4.B a hexadecimális \$8000, az XVM4.C a \$C000 címre töltődik be és SYS 8\*4096 ill. SYS 12\*4096 utasítással indítható. A parancsok a gépikódu monitornál szokásosak, de a szintaxis megismerése végett röviden összefoglaljuk azokat. A címek mindenütt négyjegyű hexadecimális számokat jelölnek a bevezető \$-jel elhagyásával.

Az egyetlen kivétel az azonnali assembler nullás lapu címzése, amit kétjegyű hexadecimális cím jelez.

A parancsok:

R - A regiszterek tartalmának kiírása. A képernyőre kiírja PC, P, A, X, Y regisztereket és a veremmutatót. A kiírt sor elején lévő ";" tulajdonképpen módosító parancs, ezért ha a sort megváltoztatjuk és RETURN-t nyomunk meg, a regiszterek értéke módosul.



- M - Memóriatartalom kiíratása. A parancs utáni első a kezdőcím, az elhagyható második a végcím. A kiírás hexadecimális formájú, soronként 8 byte kerül listázásra. A kiírt sorok elején lévő ">" itt is módosító parancsként működik, mint R-nél.
- I - Memória kiíratás karakteresen is. Mint az M parancs, de a byte-ok karakter-megfelelőjét is kiírja soronként.
- D - Visszafordítás (disassemblálás) a megadott tartományban. A disassembler által kilistázott utasítások megfelelnek az azonnali assembler szintaxisának, ennek megfelelően megvan a lehetőség a képernyőszerkesztő segítségével az utasítások helyben történő módosítására. Vigyázat: Az utasítások hosszabbra cseréjével a mögötte álló utasítások átíródnak, jelentésük megváltozik! Rövidebb utasításra való csere esetén a felszabadult byte-okat töltjük fel gondosan NOP utasításokkal, hasonló meggon-  
dolásból!
- A listázó (M, I, D) parancsok által teleírt képernyő a cursor-mozgató billentyű használatával fel-le gördíthető a parancsban megadott határokon túl is (kár részletesebben elmagyarázni, tessék kipróbálni, nagyon hasznos lehetőség).
- A - Azonnali assembler. Utasítás-szintaxisa megegyezik az assemblerével, de természetesen címkék, direktívák, makrók nem használhatók. Igen rövid, néhány utasításos programok kipróbálására, illetve programjavításra tanácsos csak használni.
- G - Program indítás. A mögötte álló címre adja a vezérlést. Ha a cím elmarad, a programszámláló aktuális értékétől indul tovább (R parancs!).
- F - Memória feltöltés. Az első címtől a másodikig a harmadik paraméterként megadott értékkel tölt fel minden byte-ot. Pl. az
- ```
F C000 C100 00
```
- parancs \$C000-tól \$C100-ig nullázza a memóriát.
- T - Memóriaterület áthelyezése. A paraméterek jelentése:
- |          |   |                                  |
|----------|---|----------------------------------|
| első     | - | az átmásolandó terület kezdőcíme |
| második  | - | az átmásolandó terület végcíme   |
| harmadik | - | a fogadó terület kezdőcíme       |
- N - Program áthelyezése. Az első két paraméter egy memóriaterületet jelöl ki, amelyen a parancs hatásos lesz. A harmadik paraméter értéke (offset) hozzáadódik minden hárombyte-os (abszolút és in-

direkt címzés módú) utasítás operandusához úgy, hogy a 16 bitről esetleg tulcsorduló eredmény utolsó 16 bitje íródik csak be természetesen a memóriába. Az utasítás hatása korlátozható a negyedik és ötödik paraméterrel, amelyek azt az átkódolás előtti címtartományt jelölik ki, ahova az átkódolandó operandusok mutattak. A parancs jól kiegészíti a T parancs hatását, ha programról van szó. Mindazonáltal igen könnyű vele programot tönkretenni, különösen, ha a programutasítások között adatok is vannak, mint általában. Ilyenkor néha egész meglepő módon átalakulhat a program. Ha az assembly forráskód is rendelkezésünkre áll, különösen hosszabb program esetén, tanácsosabb a programot újrafordítani a megfelelő helyre.

- H - Byte-sorozat keresése. Az első és a második paraméterek által meghatározott címek tartományában megkeresi a harmadik, negyedik, stb. byte-ok egymást követő előfordulásait és ezek kezdőcímét kiírja. A byte-fűző aposztróffal bevezetett ASCII karakterláncként is megadható. Pl. a

```
H 0801 0900 'ALFA
```

parancs az "ALFA" karakter-sorozat előfordulásainak kezdőcímét adja meg \$801 és \$900 között.

A fennmaradó két parancs a programok töltését és mentését végzi:

- L - Az első paraméter (idézőjelek közötti string) által megjelölt programfile-t a második paraméter által meghatározott perifériáról ( 01 - kazetta; 08, 09 - lemez) az eredeti helyére tölti.
- S - Az első két paraméter jelentése megegyezik L-ével. A mentés a harmadik és a negyedik paraméter által megadott címtartományra vonatkozik. A negyedik paraméter az első olyan byte címe, ami már nem lesz kimentve (végcím+1).

Az előzőekben megismert fejlesztői rendszer mikrogépes rendszerek átlagos, ugyszólván klasszikus eszközének tekinthető, amely alkalmas a komoly fejlesztő tevékenység támogatására, de a makrólehetőségen kívül egyéb csemegével nemigen büszkélkedhet.

### 4.3 PROFIMAT

A PROFIMAT, amelyet az NSZK-beli DATA BECKER cég forgalmaz, szintén rendszernek mondható, bár mindössze két, lazán kapcsolódó programterméket tartalmaz, a PROFI-MON monitorprogramot és a PROFI-ASS assemblert. A monitor kísértetiesen hasonlít a 4.2 fejezetben ismertetett fejlesztői rendszer monitorjára, úgyhogy csak a különbségeket írjuk le pár szóban.

A monitort a LOAD "PROFI-MON 64",8,1 paranccsal tölthetjük be és SYS 12\*4096-tal aktivizálhatjuk. Hiányzik az A és az I parancs, utóbbi funkcióját az M teljesíti. Az A, vagyis az azonnali assembler hiánya bántó hiba. Egyebekben minden parancs és azok szintaxisa megegyezik az előző fejezetben megismert monitoréval.

Egy kicsivel több figyelmet érdemel az **assembler**, ha nem is különös erényei, de elterjedtsége miatt. Szabályos szimbólikus assemblerről van szó, néhány eredeti, bár nem túl hasznos bővítéssel.

Az **utasítások szintaxisa** szinte megegyezik az eddig ismertetett assemblerekkel. A kifejezések lehetőségei igen kibővültek, a következő műveleti jelek alkalmazhatók:

- + összeadás,
- kivonás,
- \* szorzás,
- ! logikai vagy művelet,
- & logikai és művelet,
- ^ logikai kizáró vagy művelet,
- > nagyobb helyiértékű byte (egyoperandusos művelet)
- < kisebb helyiértékű byte (egyoperandusos művelet).

Az egybyte-os értékek negatív számként is írhatók (-1-től -128-ig), ezek kettes komplementes kódú alakjukkal helyettesítődnek.

A **direktívák** köre is kibővült a már más assemblerekben megismertekhez képest. Változatlan jelentéssel használható a

```
.BYTE  
.WORD  
.FILE  
.END
```

direktíva. Érdekes programozási lehetőségeket biztosít az

```
.IF  
.GOTO
```

direktívák alkalmazása. Segítségükkel feltételes fordítást végezhetünk, pl. előre megtervezhetjük a tesztelést szolgáló részek későbbi kihagyását a programból. Az .IF direktíva után logikai kifejezés, majd az egyébként .IF nélkül is használható .GOTO következik. A .GOTO után tetszőleges programbeli sorszám állhat.

Mielőtt tovább folytatnánk a direktívák ismertetését, foglalkoznunk kell a PROFI-ASS egy érdekes sajátosságával, amihez további direktívák kapcsolódnak. A PROFI-ASS tervezői igen élénk fordításideji tevékenységet képzeltek el és igyekeznek támogatni. Bár itt nincsenek makró-lehetőségek, mégis nagyon fontos, hogy élesen megkülönböztessük a fordításideji tevékenységeket a lefordított program futása közben lejátszódóktól, még akkor is, ha ez - mint jelen esetben - erősen összefonódik. A PROFIMAT assemblerre felkészült arra, hogy hívását (ami egy BASIC SYS utasítás) programból adjuk ki és forrásszövegét is ugyanott a "BASIC" programban találja. Ezáltal a BASIC nyelvet, mint egy nagygépes operációs rendszer munkavezérlő (job-control) nyelvét használva előre megtervezhetjük a fordítás, tesztelés munkáját. A programban talált assembly utasításokat lefordítja az assembler és a nekik megfelelő gépikódot közvetlenül a memóriába tölti a HELP+-hoz hasonlóan, a direktívákat azonnal végrehajtja, BASIC utasításra találva pedig visszaadja a vezérlést a BASIC interpreternek. Nézzük, milyen direktívák segítik elő ezt a működésmódot:

- <- - Az assembly programban egy szimbólum csak egyszer szerepelhet a címke mezőben, ezzel értéket kap és a következőben ezzel az értékkel helyettesítődik minden előfordulása. A <- szimbólum (értékadás) lehetőséget ad arra, hogy a szimbólumnak nem ilyen egyszeri, hanem szabadon változtatható értéket adjunk s ezáltal a fordítás vezérlésére használhasuk fel. A hatás nagyon hasonló a makróassembler-nél hiányolt makróváltozók használatához.
- Egy példa:

```
80  CIM1=$33C
90  CIM2=$C000
100 MUTATO<-0
110 LDA  CIM1+MUTATO
120 STA  CIM2+MUTATO
130 MUTATO<-MUTATO+5
140 .IF MUTATO<16 ;.GOTO 110
```

A fenti programrészlet fordításideji ciklust szervez. A futó programban hatása lehet, hogy hasonló egy futásideji cikluséhoz, de gyökeresen másképp valósítja azt meg. Megfelel a hagyományosan (.IF nélkül és változó nélkül) irt követ-

kező programnak, vagyis ugyanazt a tárgykódot eredményezi:

```
80  CIM1=$33C
90  CIM2=$C000
100 LDA  CIM1
110 STA  CIM2
120 LDA  CIM1+5
130 STA  CIM2+5
140 LDA  CIM1+10
150 STA  CIM2+10
160 LDA  CIM1+15
170 STA  CIM2+15
```

Ugyanezt a feladatot futásideji ciklussal hasonlóképpen (de nem pontosan ugyanugy és más tárgykóddal) a következő programrészlettel oldhatjuk meg:

```
80          CIM1=$33C
90          CIM2=$C000
100         LDX  #0
110  CIKLUS  LDA  CIM1,X
120         STA  CIM2,X
130         TXA
140         CLC
150         ADC  #5
160         CMP  #16
170         BMI  VEGE
180         TAX
190         JMP  CIKLUS
200  VEGE
```

A példák értelmezését és összehasonlítását az olvasóra bizzuk.

A következő direktívák a szimbólumtábla kezelést végzik. A szimbólumtábla az assembler által felépített táblázat, amely a szimbólumokat és a hozzárendelt értékeket tartalmazza.

.SST <egységszám>,<másodlagos cim>,"<filenév>"  
A teljes szimbólumtáblát a megadott egységre, általában egy szekvenciális lemezfile-ba menti, későbbi felhasználás céljából.

.LST <egységcim>,<másodlagos cim>,"<filenév>"  
Az előző direktíva ellentéte, a korábban .SST-vel kimentett szimbólumtáblát a memóriába tölti.

A két direktíva alkalmazásával lehetőség nyílik nagy programok részleteinek külön fordítására, de az egyes részek között, a kapcsolat megteremtésére (hasonlóan a szerkesztő-programok - linkage editorok - működéséhez). A korábban fordított programban szereplő szimbólumokra hivatkozhatunk a később fordított programban, ha az első fordítás során a szimbólumtáblát kimentjük, a második fordítás első lépéseként pedig visszatöltjük.

Végül ismerkedjünk meg az .OPT (opciók) direktíva lehetséges paramétereivel. A direktíva után vesszővel elválasztva akárhány paraméter állhat.

- P - Hatására az assembler formázott forráslistát készít a képernyőn. Ha nincs érvényben, akkor is listázásra kerülnek a hibás sorok és a .FILE direktívát tartalmazó sorok.

P<filesorszám>

- Hatására a lista az assembler elindítása előtt OPEN utasítással megnyitott file-ra - leggyakrabban printerre - kerül.

- 0 - A tárgykód egy speciális memóriaterületre - a BASIC tömbök helyére - kerül, s onnan tetszés szerint kimenthető

- 00 - A tárgykód az eredeti helyére kerül, a programban magában megadott tárolóterületre. Ha ez a terület valamilyen okból foglalt, az előző módszert (0) kell használni.

0<filesorszám>

- A tárgykód az assembler elindítása előtt OPEN utasítással megnyitott file-ra kerül. A kazettás egység itt nem alkalmazható.

- N - Az utána felsorolt paraméterek hatását törli. Ha egymagában áll, valamennyi opciót hatástalanítja.

**Az assembler használatáról** is szólnak néhány szót!

Az assemblert a LOAD "PROFI-ASS 64",8,1 utasítással tölthetjük be lemezről, majd az akár programból, akár közvetlen utasításként kiadott SYS 9\*4096-tal indíthatjuk el.

Az assembler a BASIC terület végén helyezkedik el, mintegy 34 Kbyte szabad területet hagyva a program és a szimbólumtábla számára.

Egy tipikus PROFIMAT program:

```
10 OPEN 5,8,5,"TARGYKOD";REM EBBE A FILEBA FOG FORDITANI
20 OPEN 3,4;REM NYOMTATO MEGNYITASA
30 SYS 9*4096 ;REM ASS HIVASA
40 .OPT 05,P3; TARGYKOD 5-05, LISTA 1-ES FILERA
50 .LST 8,8,"KONYVTAR"; SZIMBOLUMTABLA TOLTESE
```

```

60  *=FOPRV; FORDITAS A FORPROGRAM UTANRA
70  KOD1=$C1; KODTABLA
80  KOD2=$C9
90  ELLEN    LDA    PARAM
100          CMP    #KOD1
110          BEQ    K1
120          CMP    #KOD2
130          BEQ    K2
140  HIBAS   LDA    #$FF
150          RTS
160  K1      LDA    #1
170          RTS
180  K2      LDA    #2
190          RTS
200  ELLENV  *=
210  .SST    B,B," :KONYVTAR"
220  .END
230  CLOSE   3:CLOSE   5: PRINT "VEGE A FORDITASNAK"

```

Néhány dolgot érdemes megfigyelni a fenti programban. Az 50-es sorban lévő direktívával betöltjük a korábban kimentett szimbólumtáblát a memóriába. Ezáltal lehetőségünk nyílik a jelen programban nem – de feltételezhetően a korábban kimentett szimbólumtáblában – szereplő PARAM és FOPRV szimbólumokra hivatkozni. A 210-es sorban a fenti KONYVTAR nevű file-t felülírjuk a jelen program fordítása során kiegészített szimbólumtáblával, ezáltal egy következő fordítás már az itt megadott szimbólumokat (pl. a szubrutin kezdőcímet) használhatja.

**Osszefoglalva** a PROFIMAT által nyújtott szolgáltatásokat (amelyek közül a legbonyolultabbakat és legnehezebbeket felhasználhatók nem is tárgyaltuk) megállapíthatjuk, hogy érdekes lehetőségeket felvonultató fejlesztői rendszerrel van szó, amely azonban nem éppen a strukturált programozás és mérnöki tevékenység típusú szervezett programfejlesztés támogatására szolgál, inkább az érdeklődő amatőrök számára készült.

#### 4.4 SM-KIT 64

Az SM-KIT 64 lényegében a HELF+-hoz hasonló, de annál valamivel szélesebb lehetőségeket biztosító alapsoftware. Három részből áll: a BASIC programozást és programtesztelést, a gépkódu programozást és végül a VC 1541-es lemezegység processzorának programozását támogató részekből. A következőkben természetesen csak a témánkat, az assembly programozást érintő részletekről szolunk. Érdekes módon, míg a rendszer az eddig ismertetett eszközöknél valamivel hatékonyabban támogatja a programok tesztelését, kipróbálását, az assembly programok fordítását szinte egyáltalán nem, mivel mindössze egy debuggerhez illő azonnali assemblert tartalmaz. Természetesen egy tetszőleges assembler által lefordított program már kezelhető az SM-KIT segítségével.

Az SM-KIT 64 három része egyetlen programot kepez, amelyet a programlemezről a LOAD "\*",8,1 utasítással tolhatunk a memóriába. Ezután az SM-KIT "."-tal kezdődő parancsai használhatók, amelyeket az alábbiakban ismertetünk.

.<cim> - disassembler funkció, amely egy képernyőnyi outputot ad, majd RETURN-re újabbat. RUN/STOP-pal leállítható. A disassemblálás kezdőcímét a <cim> adja meg.

.<cim><assembly utasítás>

- lehetőséget ad egyetlen utasítás lefordítására a <cim> által jelölt memóriaterületre. A címek megadása alapállapotban decimálisan történik. Ez hexadecimálisra változtatható .H megadásával, ami decimálisra változtatható .\$-ral. A szám után írt H vagy \$ egyszeruen hexadecimális - decimális ill. decimális - hexadecimális átváltást tesz lehetővé a kijelzési mód megtartásával.

.<cim>B

- A <cim>-től kezdve decimális modban 5, hexadecimálisban 8 byte-ot ír ki. RETURN megnyomásával újabb sor kerül kiírásra.

.<cim>A

- A <cim> által mutatott és az azt követő byte-on elhelyezkedő vektor (alsó byte-felső byte) értéke kerül kiírásra. Különösen decimális modban hasznos.

.<cim> - (aposztróf) A <cim> által mutatott memóriaterület első 8 byte-jának kiírása karakteres formában a fenti utasítások mind alkalmasak arra, hogy segítségükkel a memória tartalmát ne kiirassuk, hanem megváltoztassuk. Ez akkor következik be, ha a kód (B, A, , disassemblernél semmi) után annak megfelelő formátumu érték áll. 58



.T<cim1>,<cim2>,<cim3>

- A <cim1>-től a <cim2>-1 terjedő tartományt átmásolja a <cim3>-tól kezdődő munkaterületre (transzfer)

.P<cim1>,<cim2>

- segítségével egy programterületet jelölhetünk ki a következő I utasítás számára. Paraméter nélkül megadva kijelzi az aktuális programterületet.

.I<cim1>,<eltolás>

- A <cim1>-en kezdődő programrészletet - a végét .P-vel definiáltuk - a megadott értékkel eltolja. A JMP és JSR utasításokat átcimzi, a küldő területet NOP utasításokkal tölti fel, a .P által kijelölt programterületet kiterjeszti ill. leszűkíti negatív eltolás esetén (azaz a végcim az eltoló program végcime lesz, a kezdőcim az eredeti marad).

.F(<cim>),<kód>:<operandus>

- Bizonyos utasítások kereséset végzi a memóriában. A keresést egy, a <cim> által mutatott táblázat szerint végzi, amely 2-2 címet tartalmaz, egy kezdő- és egy végcímét minden programblokkra. A táblázatot két bináris 0 byte zárja le. A keresett utasításokat utasításkód és/vagy operandusérték szerint azonosíthatjuk. A <kód> vagy egy utasítás gépkódja, vagy a következő szimbólumok valamelyike:

S        - minden olyan abszolút címzésmodu, amely egy cím tartalmát megváltoztatja (pl. STA, INC, ASL, SLB.),

SI        - ugyanez, de abszolút, indexelt címzésmodra,

SII      - ugyanez, de indexelt indirekt és indirekt indexelt címzésmodra,

L,LI,LI1

- mint fent, de olyan utasításokra, amelyek egy memóriabyte-ot regiszterbe töltenek vagy műveletet hajtunk végre velük (pl. LDA, EOR, ADC stb.), tehát egy regiszter tartalmát megváltoztatják.

Ha a :<operandus>-t is megadjuk, a keresés csak az ilyen operandusú utasításokat listázza ki disassembler formában. Az utasítás igen alkalmas programhibák felkutatására.

A következőkben ismertetésre kerülő utasítások a gépkódu programok nyomkövetését segítik elő. Segítsejükkel végrehajtás közben vizsgálhatjuk a programok hatását. Igen hasznos eszközt jelentenek még annak fenntartásával is, hogy mivel a programfutas idoviszonyait alapvetően megváltoztatják, bizonyos esetekben használhatallanok ill. hamis eredményt. A lépésenkénti végrehajtáshoz az SM-KIT-nek szüksége van egy olyan területre, amelyet a 6510-es mikroprocesszor esetében kitüntetett fontosságú 0-ás és 1-es lap helyett használ. Ezt tetszés szerinti memóriaterületen kijelölhetjük a

.><cim> utasítással. A munkaterület hossza 768 byte.

A fenti utasításra a monitor egy

```
.> AAAXXXYYYSSS--++----  
..  a  x  y  s  nv  bdizc
```

alakú választ ad, ahol a fenti sor az aláírt regiszterek és jelzők értékét mutatja. Az első sor végére tetszőleges címet írva a monitor végrehajtja a címen levő utasítást, a fentieknek megfelelően kijelzi a regiszter-tartalmakat, a következő utasítás címét és magát az utasítást disassembler formában. A RETURN megnyomásával a folyamat a következő utasítással megismétlődik.

Ha a sor végén, a <cim> helyen -<cim> formában egy végcím-kifejezést írunk, a program a megadott címig nyomozás nélkül fut, majd ott a szokásos regiszterkijelzéssel megszakad. Természetesen, ha a program nem érinti a megadott címet, a vezérlés kicsuszik a kezünkől, a programnál marad.

A már kipróbált - vagy például ROM-ból hívott - szubrutinok nyomozása feleslegesen lassítaná a nyomozást. Ezért lehetőség van arra, hogy a nyomozást egy JSR utasításnál megszakítsuk és az RTS végrehajtása után automatikusan folytassuk. Ha a lépésenkénti végrehajtás során a következő utasítás egy JSR - szubrutinra ugrásutasítás, a következő sor elején egy "-R" szimbólum megadása a fenti eljárást váltja ki a nyomozásban, tehát az öt logikailag zárt RTS-ig a lépésenkénti végrehajtás felfüggesztődik.

Az SM-KIT egy érdekes és igen hasznos tulajdonsága, hogy a lépésenkénti végrehajtás során a memóriakiírások újra meg újra végrehajtásra kerülnek. Ennek segítségével a nyomozás során figyelemmel kísérhetjük a kiválasztott memóriamezők változását. A nyomozás alatt a képernyő tetején tetszés szerinti memóriakiírásokat kérünk (A, B, '), majd a képernyő alsó részén elindítjuk a lépésenkénti végrehajtást. Minden utasítás után a memóriakiírások ismételten végrehajtnak, így a képernyőn mindig az aktuális memóriatartalmat láthatjuk.

Az utolsó nyomozási eszköz software megszakítási pont elhelyezését biztosítja a programban. A .B<cim> utasítással egy csapdát helyezünk el a megadott címen, a program futása megszakad, a regisztereket és a jelzőket kilistázza a monitor, majd újabb parancsra vár.

Az SM-KIT utolsó itt ismertetésre kerülő lehetősége nem a programok kipróbálását szolgálja. Az igazi helye az 5.1 fejezetben lenne, ahol többek között azzal is foglalkozni fogunk, hogy hol helyezzünk el olyan rövid assembly rutinokat, amelyeket BASIC nyelvű programból akarunk hívni. Erre egy - elég bonyolult, de sok szempontból praktikus - lehetőség magában a programban, egy BASIC sorban, a REM után elhelyezni. Az ilyen szubrutint szokás **REM-rutinnak** is hívni. Az ilyen rutin készítését segítik a következő SM-KIT parancsok:

.Z <sorszám> - A megadott sorszámú BASIC sor kezdőcímét kiírja a képernyőre. Ha a sor nem létezik, az első nagyobb sorszámú sor adatai kerülnek kiírásra sorszámosítva. Ha a sorszám nagyobb bármelyik sorszámnál a BASIC-programban 64000-es sorszám és két 0 kerül kiírásra.

.TP,Z <sorszám> - A .P által előzőleg megadott programot a BASIC programba illeszti a megadott sorszámú sorba, mint REM-rutint. Így pl. a

.P 828,833

.TP,Z 10

parancssorozat a 828-tól 833-ig terjedő címeken lévő programot REM-rutinná formálva beilleszti a tárban lévő BASIC-program 10-es számú soraként.

A REM-rutinnak kötelezően 0-val kell végződnie (BASIC-előírásból következően)

Az így létrehozható REM-rutinokkal kapcsolatos legfontosabb korlátozások:

- hossza legfeljebb 250 byte lehet,
- nem tartalmazhat önmagára hivatkozó JMP vagy JSR utasítást,
- felhívása kicsit körülményes, mivel a BASIC program szerkesztésével helye változhat a tárban, hacsak nem írjuk azokat a program legelejére.

Az SM-KIT-et, mint az assembly programozást segítő eszközt **összefoglalva** megállapítható, hogy minden eddig ismertetett-nél szélesebb körű lehetőséget biztosít gépiódu programok bejáratására, elemzésére, egy jóminőségű debuggert nyújt. Hibájaként róható fel viszont, hogy nem ad eszközöket a programok létrehozására, nem tartalmaz a monitor-debuggerrel egységben működő assemblert, ezért csak jóminőségű assembler editor környezettel kiegészítve ajánlott - úgy viszont különösen - az assembly fejlesztői tevékenységhez.

#### 4.5 PTD interaktív debugger

Az ebben az alfejezetben ismertetésre kerülő debugger (PTD 6502/6510 DEBUGGER, PTERODACTYL SOFTWARE, CA USA) egy sokkal magasabb rendű, ennél fogva bonyolultabb eszközt jelent, mint az eddigiek. Ezért talán nem érdektelen részanni egy kicsit terjedelmesebb szövegrészt a megismerésére, mint az eddig ismertettekkel tettük. Már előljáróban megemlítjük, hogy ez a program nem kötődik kifejezetten a Commodore 64-hez – mint könyvünk zöme sem –, hanem minden 6502/6510 mikroprocesszorral szerelt személyi számítógéphez, így pl. az Apple II sorozathoz és a nagyobb Commodore modellekhez is használható.

Talán nem árt már előljáróban felsorolni a PTD debugger **jellemzőit**, amelyeket a továbbiakban részletesen is tárgyalunk. Természetesen rendelkezik a minden debuggerre szinte kötelező minimális utasítás-csoporttal:

- memória-dump hexadecimális és ASCII formátumban,
- memória-módosítás lehetősége,
- disassembler funkció
- gépkódu program lépésenkénti végrehajtása,
- tetszés szerinti információk nyomtaton is megjeleníthetők,
- azonnali assembler.

Ezenkívül a következő, egyáltalán nem általános funkciókat is teljesíti:

- utasítások batch típusu végrehajtása, ami által lehetőség nyílik tesztágy kialakítására,
- szimbólikus címzés lehetősége a bejáratás során,
- beépített, rezidens szimbólikus assembler,
- öndokumentáló, érthető parancsnevek, amelyek a gyakorlott felhasználó által lerövidíthetők,
- értelmes, angol nyelvű hibaüzenetek,
- kiterjesztett kifejezésfogalom,
- különböző számrendszerbeli számok szabad keverése,
- a programok részenkénti nyomkövetési lehetősége,
- kód-foltozás (patching – jelentését ld. később),
- képernyőfelosztás a nyomozás alatt felhasználói és nyomozási ablakra,
- a végrehajtott gépi ciklusok számlálása, amely elősegíti a kód utólagos hangolását,
- a joystick használata a nyomozás vezérléséhez.

#### 4.5.1 A debugger betöltése

A lemezről LOAD "LOADER.OBJ",8,1 utasítással tölthetjük be a programot – illetőleg a töltőprogramot –, majd SYS 52000-rel indíthatjuk. A program ezután kéri a debugger számára szükséges 28 Kbyte-nyi terület kijelölését.

A választható opciók:

- T        - felső memória (\$6000-\$D000)
- B        - alsó memória (\$800-\$7800)
- <érték> - kezdőcím (tetszőleges \$800-nál nagyobb érték)

Utóbbi esetben gondoskodni kell a képernyő-memória számára külön 2 Kbyte-nyi terület kijelöléséről. Erre a területre azért van szükség, mert a debugger minden beavatkozás előtt menti a képernyő-memória és a szín-memória területét, hogy a felhasználói programot a legkevesebb befolyásolja. T és B esetén a mentési pufferek kijelölése automatikus. A debugger futása alatt a felhasználói képernyőre az F3, a debugger képernyőre az F1 megnyomásával kapcsolhatunk át.

A debugger betöltése után a verem értékes részének kijelzésével jelentkezik a képernyőn.

A debuggerből az END paranccsal léphetünk ki. A vezérlés visszakerül a hívóprogramhoz, általában a BASIC-parancs módhoz.

#### 4.5.2 Alapfogalmak

A debugger által elfogadott számok lehetnek decimálisak, hexadecimálisak (\$-ral bevezetve), binárisak (%), negatív decimálisak (-). A karaktर्सorozatok tetszős szerint aposztrofok vagy idézőjelek közé zárhatjuk. A kifejezések +, -, \*, /, %OR, %AND és %XOR műveleti jeleket, valamint zárójeleket tartalmazhatnak a szokásos értelmezésben.

A debugger minden parancs végrehajtása után a sor elején vessző kiírásával (prompttal) jelzi, hogy kész a következő utasítás fogadására.

A ? utasítást használhatjuk – a BASIC-hez hasonlóan – kifejezések értékének kinyomtatására. A képernyőn megjelenik a kifejezés értékének decimális és hexadecimális formája egyaránt. Néhány billentyű speciális jelentéssel bír:

- RETURN        egymagában megismétli az előző parancsot,
- F1            átkapcsol a debugger-képernyőre,
- F3            átkapcsol a felhasználói képernyőre,
- RESTORE       "melegindítja" (ld. később) a debuggert,
- RUN           hatása megegyezik a RUN paranccsal (ld. később)

### 4.5.3 Program-módu végrehajtás

A FTD debugger parancsait a BASIC-hez hasonlóan kétféle módban lehet végrehajtani: parancs vagy **azonnali módban** (immediate mode) és **program-módban** (deferred mode).

Előbbi esetben az utasítás a rendszer promptja után rögtön begépelendő és azt a debugger azonnal végre is hajtja. A program-módban viszont a debugger utasításait a BASIC programok mintájára sorszámmal bevezetve gépeljük be, amelyeket a rendszer ilyenkor csak tudomásul vesz. A végrehajtás a RUN parancs hatására történik meg. A debugger 1 Kbyte-ot tart fenn alaphelyzetben a program-módu utasítások tárolására. Ez rendszerint elegendő, de szükség esetén ki is terjeszthető.

A program-módu utasítások kezeléséhez négy parancs áll rendelkezésre. A LIST, RUN, NEW parancsok kezelése és hatása megegyezik a BASIC megfelelőjével. A sorok törlését sorszámmal egymagában való megadásával, javítását az EDIT parancssal végezhetjük.

### 4.5.4 Szimbólumok

A szimbólumok lehetnek :

- változók,
- regiszterek és jelzők nevei.

A **változók** legfeljebb 15 karakteresek lehetnek, numerikus vagy karakteres típusúak. A numerikus változók kétbyte-os egész számokat tartalmazhatnak tetszőleges számrendszerbeli értékadással. A karakteres változók neve kötelezően \$ jellel végződik. Csak program-módban kaphatnak értéket, amely legfeljebb 255 byte hosszú lehet.

A **fenntartott nevek** nem adhatók meg változónévként, ezek az assembly utasítások, parancsok, valamint a regiszterjelzők nevei.

Utóbbiak:

- |   |    |   |                                |
|---|----|---|--------------------------------|
| - | A  | - | akkumulátor                    |
| - | X  | - | indexregiszter                 |
| - | Y  | - | indexregiszter                 |
| - | PC | - | utasításszámláló               |
| - | P  | - | állapotregiszter               |
| - | S  | - | verem-mutató                   |
| - | %C | - | átvitel-jelző                  |
| - | %D | - | decimális mód jelző            |
| - | %I | - | megszakítás-érvényesítés jelző |
| - | %N | - | negatív érték jelző            |
| - | %V | - | tulcsordulás jelző             |
| - | %Z | - | nulla jelző                    |

A szimbólumok értékadó utasításokkal nyerhetnek értéket, amely a BASIC-ben megszokott formájú. Vonatkozik ez a regiszterekre és a jelzőkre is! Így az

X=A

helyes utasítás (hatása ekvivalens TAX gépi utasításával). A jelzők értéke magas lesz nem nulla kifejezéssel történő értékadással, ellenkező esetben alacsony lesz. A kifejezésekben a regiszterek és jelzők szabadon használhatók. Azok értéke természetesen mindig a felhasználói program által "látott" és nem a debugger által pillanatnyilag használt valódi regiszterekkel és jelzőkkel egyezik meg.

#### 4.5.5 DOS kiegészítés

A VC 1541 lemezegység vezérlésére a debugger kiegészítő parancsokat nyújt.

A DOS "\$" parancs a lemezkatalógust listázza a képernyőn. A DOS parancs után idézőjelek között tetszőleges, a disk-vezérlőnek szóló parancsot kiadhatunk, pl. DOS "S:T\*" törli a T betűvel kezdődő nevű file-okat.

#### **Programok töltésére a**

LOAD "<filenév>" "<kezdőcim>"

alaku utasítást használhatjuk. A parancs LD-re rövidíthető, a <kezdőcim> elhagyható, ilyenkor a lemezen tárolt kezdőcimre töltődik be a program.

#### **Programok mentésére a**

SAVE "<filenév>"<kezdőcim><végcim+1>

alaku parancs szolgál. A SAVE SV-nek is rövidíthető, a cimek megadása kötelező.

Debugger-utasításokat tartalmazó **program-file-ok** **töltése** az **eltérő**

LOADP "<filenév>"

utasítással (LOADP helyett LDP elég), **mentése** a

SAVEP "<filenév>"

alaku (SAVEP helyett SVP írható) utasítással történik. LOAD és LOADP utasítások végrehajtása után a debugger kiírja a program által elfoglalt utolsó byte címét.

#### 4.5.6 Memóriakezelés

A DUMP <kezdőcim><hossz> alaku parancs szolgál a memória egy részének hexadecimális és ASCII formájú kiírására. A <hossz> elhagyható, ekkor egy sor – 8 byte – kerül kiírásra.

Emellett használható a BASIC PEEK függvényhez és POKE utasításhoz igen hasonló két lehetőség a memória vizsgálatára és átírására. Felhasználásuk nyilván program-módban előnyös. A szintaxis megegyezik a BASIC megfelelőekkel, de szabadabb, a zárójel ill. a vessző az utasításban szóközzel helyettesíthető.

A <cim> alaku utasítással egy kétbyte-os értéket írathatunk ki. Az utasítás természetesen figyelembe veszi a 6510 szokásos alacsony helyiértékű byte – magas helyiértékű byte sorrendjét.

A memóriakezelés másik oldala a disassembler és assembler funkciók, tehát programok listázása a memóriából, illetve programrészletek (vagy teljes programok) beillesztése.

A disassemblert a

DISASM <kezdőcim><sorok száma><cim>

alaku utasítással aktivizálhatjuk. Valamennyi paraméter elhagyható. A paraméterek és alapértékük (zárójelben):

|               |   |                                                                                                                                                                                                                 |
|---------------|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <kezdőcim>    | - | a visszafordítandó program kezdőcíme (az utasításszámláló állása)                                                                                                                                               |
| <sorok száma> | - | a listázandó utasítások száma (15)                                                                                                                                                                              |
| <cim>         | - | a disassembler ide helyezi (ha megadtuk) a visszafordított programot ASCII-assembly forráskód formájában. Innen később SAVE paranccsal kimenthető lemezre. A terület végcímét DUMP utasítással kereshetjük meg. |

A PTD debugger, mint már említettük, egy jóminőségű, kétmenetes szimbólikus assemblert tartalmaz, amely azonban csak aránylag rövid, mintegy 100 soros programok fordítására alkalmas. Fő hivatása nem is az önálló programozás, hanem a kód-foltozás (ld. később) támogatása. Az assembler által elfogadott nyelv szintaxisa megegyezik a szokványos formátummal. Egyetlen korlátozás, hogy program-módban csak megjegyzést tartalmazó sorok nem megengedettek.

Az assembler tulajdonképpen egy külön alrendszert képez a debuggeren belül. Ennek legfontosabb következménye az, hogy az assembler aktivizálása után debugger parancsok nem adhatók ki, egészen a debuggerhez való visszatérést kiváltó RET assembler direktíváig.



Az assemblert kétféleképpen használhatjuk parancs és program-módban.

**Parancs-módban** az

ASM <kezdőcím>

alaku parancs után a debugger megszokott vessző ("") prompt-ját felkiáltójel ("!") váltja fel, ez utal arra, hogy itt assembly utasításokat vár el a program az addigi debugger parancsok helyett. Az assembly sorokat sorszám nélkül (!) adhatjuk be. A sorszámmal beírt sorok nem kerülnek fordításra, hanem a programterületen tárolódnak. A szokásos assembly utasítások megengedettek, de az operandus mezőben a debugger alatt definiált változók használhatók, értékük az assemblerben is elérhető. Az ASM parancsban megadott kezdőcím határozza meg azt, hogy hova kerül a lefordított program a memóriában. Ha ez elmarad, a programszámláló aktuális értékétől kezdődik a kód lerakása. A fordítás befejezése után a programszámláló az első szabad helyre mutat. Az assembly utasításokon kívül két direktívát tartalmazhat a program, a DUP és a RET direktívákat. Használatukra a kód-foltozás tárgyalásánál térünk ki, de már most megemlítjük, hogy a paraméter nélkül írt RET egyszerűen lezárja az assembly fordítást és visszaadja a vezérlést a debuggernek. Lássunk egy rövid példaprogramot az assembler parancs-módu használatára:

```
,ASM $C000
!SZAMX NOP
!SZY TYA
! RTS
!BELEP STX SZAMX
! CPY SZAMX
! BMI SZY
! TXA
! RTS
!RET
```

A programhoz nem árt egy kis magyarázat.

Az első sor felhívja az assemblert és közli a program kezdőcímét (\$C000). A program feladata, hogy X és Y regiszterben lévő két szám közül eldöntse, melyik a kisebb és azt az A regiszterben visszaadja. A kicsit kitekert írásmódot az kívánta meg, hogy a PTD assemblerében parancs-módban nincs lehetőség előre hivatkozásra, minden szimbólumnak a felhasználáskor már definiálnak kell lennie. Ezért a SZAMX memóriaváltozót és az SZY programcímét előre kellett megadni. A szubrutin végrehajtása a BELEP ponton kezdődik. A záró RET visszaadja a vezérlést a debuggernek.

Sokkal hatékonyabban használható az assembler **program-módban**. Ekkor az ASM utasítást és az assembly programot egyaránt, mint egy PTD debugger program utasításait kell megadni. Ebben az esetben természetesen előre hivatkozások is használhatók a programban. Az assembler használata program-módban nagymértékben emlékeztet a PROFIMAT assembleréhez és

annak megfelelően itt is felhasználhatjuk a parancs-nyelv feltételes vezérlésátadását makrószerű lehetőségek igénybevételére. A megoldás szinte tökéletesen megegyezik az ott bemutatattal:

```
10  Z1=$14
20  MUTATO=2
30  SORSZAM=1
40  ASM $C000
50  LDA #C1
60  STA Z1+1
70  LDA #0
80  STA Z1
90  RET
100 CIKLUS
110 ASM
120 LDY #MUTATO
130 LDA #SORSZAM
140 STA (Z1),Y
150 RET
160 IF SORSZAM<3 THEN SORSZAM=SORSZAM+1:MUTATO=MUTATO+5:
165 GOTO CIKLUS
```

Bár a program két olyan lehetőséget is tartalmaz, amelyről még nem volt szó (címkét és if utasítást) jelentése így is elég világos. A ciklusszervezéssel elértük azt, hogy a középső részt (120-140 sorszám-tartomány) csak egyszer kellett leírni. A programrészlet egyébként egy, a \$C100-on elhelyezett táblázatot tölt ki. Az egymásutáni 5 byte-os bejegyzések 3. byte-jába a bejegyzés sorszámát (1-3) teszi. A programot hagyományos módon így írtuk volna:

```
10  Z1=$14
20  ASM $C000
30  LDA #$C1
40  STA Z1+1
50  LDA #0
60  STA Z1+1
70  LDY #2
80  LDA #1
90  STA (Z1),Y
100 LDY #7
110 LDA #2
120 STA (Z1),Y
130 LDY #12
140 LDA #3
150 STA (Z1),Y
160 RET
```

A két program által generált gépikód tökéletesen megegyezik. Visszatérve az előző megoldáshoz, vegyük észre, hogy az ASM utasítást a második (ciklusba ágyazott) esetben paraméter nélkül hívtuk, ennek eredményeként a fordítást oda folytatta, ahol az utolsó RET után félbeszakadt. Így a generált kód folyamatos lesz.

Az assembler a következő direktívákat ismeri:

- .BA - program kezdőcíme (\* helyett)
- .DS - byte-ok foglalása (darabszám követi)
- .BY - egy byte foglalása (kezdőértékmegadás követi)

Meg ehhez az alfejezethez sorolható, bár általános érvényű a következő parancs:

PORT <egységszám>

Hatása hasonló a BASIC OPEN és CMD utasítások együttes hatásához. PORT 4 kiadása után a felhasználói program outputjai csak a printeren, a debugger outputjai a printeren és a képernyőn jelennek meg. A képernyős kijelzési módhoz való visszatérést a PORT 0 utasítás váltja ki.

#### 4.5.7 Program-nyomkövetés

Elérkeztünk a debugger azon szolgáltatásaihoz, amelyek a legfontosabbak felhasználhatósága szempontjából. A PTD debugger széles körű lehetőséget biztosít a programok nyomkövetésére, a fő problémát leginkább az eszközök közötti változás és a parancsok néha bonyolult szintaxisának megtanulása jelenti.

A STEP utasítás segítségével a felhasználói programot lépésenként hajthatjuk végre. Az utasítás formája:

STEP<lépésszám>PC<cím> DISP/NODISP/JOY

A <lépésszám> a végrehajtandó gépiódu utasítások számát jelenti. Ha nem adjuk meg, a debugger egyetlen utasítást hajt végre, ha 0-át adunk meg, 65535-öt. Ha a DISP paramétert megadjuk, minden lépés, azaz minden végrehajtott utasítás után információkat ír ki a debugger, ha NODISP-et vagy semmit sem adunk meg, csak az utolsó utasítás után – tehát ha elértük a <lépésszám> által megadott limitet – íródnak ki információk. Ez utóbbi esetben a nyomozott program futása lényegesen gyorsabb, mintegy 1000 utasítás másodpercenként.

A PC <cím> paraméter a nyomozás kezdőcímét állítja be az utasításslámlálóba, tehát ugyanazt az eredményt nyújtja, mintha a STEP parancs kiadása előtt a

PC = <cím>

értékkadást elvégeztük volna. Elhagyása esetén a nyomozás kezdőértéke az utasításslámláló aktuális értéke.

A JOY paraméter ugyanugy kiváltja minden utasítás után a kiíratást, mint a DISP, de a 2-es számú joystick-csatlakozóba dugott joystick-kel lehetőségünk van a végrehajtás sebességét csökkenteni ill. a gomb megnyomásával teljesen felfüggeszteni. Ezáltal lehetőség nyílik a képernyőn futó információk alaposabb elemzésére.

Egy-egy végrehajtott utasítás után a DISP vagy a JOY érvényessége estén a következő sor jelenik meg:

```
<cim><utasítás>A<érték1>Y<érték>X<érték>P<érték><op><érték2>
```

ahol

```
<cim>          - a végrehajtott utasítás címe
<utasítás>    - disassemblált utasítás
<érték1>      - az A regiszter értéke hexadecimálisan és
                ASCII-ben
<érték>       - a megfelelő regiszter értéke hexadecimálisan
<op>          - a kiszámított memóriacím
<érték2>      - az ezen a címen lévő érték hexadecimálisan
                az utasítás végrehajtása előtt.
```

Az aláhuzott betűk a képernyőn reverse módban jelennek meg.

Ha a STEP végrehajtása közben bármely billentyűt megnyomjuk, a STEP végrehajtása megszakad. Ekkor egy előre deklarált változó, BRKFLG fogja tartalmazni a billentyűnek megfelelő karakter ASCII kódját. Ezt az értéket a nyomozást vezérlő programban feltételes vezérlésátadó utasításban használhatjuk fel.

A DISP/NODISP/JOY paraméterek addig maradnak érvényben, amíg egy másikat ki nem adunk, tehát nem kell megismételni azokat a következő STEP utasításban. Így pl. a következő utasítás-sorozat:

```
,STEP PC $C000 DISP
,STEP
,
,
,
```

jelentése a következő ( a magában álló promptok azt jelzik, hogy ott csak a RETURN-t nyomtuk meg, ami az utoljára végrehajtott utasítást ismétli):

- egy utasítás végrehajtása a \$C000 címről és az információk kiírása
- a következő utasítás végrehajtása kijelzéssel
- utóbbi ismételtetése.

Megjegyezzük, hogy ez a debugger leggyakoribb és legegyszerűbb használati módja.

A SUB parancs szintaxisa és a paraméterek jelentése teljesen megegyezik a STEP-ével. A különbség mindössze csak az, hogy DISP és JOY módban, ha JSR (szubrutinhívás) utasításhoz ér, a debugger megkérdezi, vajon a felhívott szubrutint gyorsan (nyomozás nélkül) akarjuk-e végrehajtani. Ha RETURN-nel válaszolunk, a szubrutin teljes sebességgel fut le. Így kikerülhetjük már biztonságosan letesztelt ill. operációs rendszer-beli szubrutinok felesleges lépésenkénti végrehajtását. NODISP esetén a végrehajtás módja semmiben sem különbözik STEP-étől (bár ezt a kijelentést majd a tartományok megismerése után pontosítjuk).

A teljes sebességgel végrehajtott szubrutinból visszatérve a végrehajtás az eredeti nyomozási feltételek szerint folyik tovább.

A következő, igen fontos eszköz a nyomozás során a felhasználói megszakítások alkalmazása. A megszakítást a programban egy BRK utasítás végrehajtása idézi elő, amelyet a BRK debugger paranccsal (és nem assembly utasítással!) célszerű elhelyezni. Ennek formátuma:

BRK <név><cím>

ahol a <cím> egy még nem használt azonosító, amellyel a megszakítási pontra szimbólikusan hivatkozhatunk, a <cím> azon utasítás első byte-jának címe, ahol a megszakítási pontot el kell helyezni. A megadott címen lévő byte-ot a debugger egy táblázatba menti és helyére egy BRK (\$00) utasítást tesz. Ha a debugger a felhasználói program teljes sebességű végrehajtása során egy BRK utasításba utkozik, a következőket végzi el:

- visszamenti a regisztereket (amelyet a BRK hatására a mikroprocesszor elmentett a verembe),
- végrehajtja azt az utasítást, amelynek a helyén a BRK van, mintha arra egy STEP DISP utasítást adtunk volna ki,
- nem menti vissza a táblázatba kimentett utasításkódot a BRK helyett,
- a BRKFLG változót 2-re állítja,
- program-módban áttér a következő utasítás végrehajtására,
- parancs-módban promptot ír ki és utasításra vár.

Program-módban való végrehajtás esetén a következő utasítás elágazást hozhat létre a BRKFLG alapján, amely jelzi a megszakítás okát. A megszakítási pontot a

BRKCLR ALL  
BRKCLR MEGSZ1

tipusu utasítások valamelyikevel lehet törölni. Az első típus valamennyi megszakítási pontot törli és a kódot helyrehozza (a kimentett utasításkódot visszamenti). A második csak a megadott névűre hatásos. Itt vesszővel elválasztva több megszakítási pont is felsorolható.

Az eddig elmondottak az ugynevezett software-megszakítást jelentették. A debugger ezenkívül lehetőséget ad az un. lassu megszakítási pont elhelyezésére is. Ezt használhatjuk a ROM-ban lévő programok nyomkövetésére, mivel itt a program módosításának lehetősége nem áll fenn. A megszakítási pontot a

SLOWBRK <név><cím>

paranccsal állíthatjuk be, ahol a paraméterek jelentése megegyezik a BRK paranccsával. Ettől az utasítástól kezdve a felhasználói program minden utasítása után megszakad és a debugger ellenőrzi, hogy elértük-e valamelyik lassu megszakítási pontot. Ennek megfelelően a program végrehajtása nagymértékben lelassul. Ha a debugger egy lassu megszakítási pontra ér

- Üzenetben közli, hogy megszakítási pontra talált,
- a BRKFLG változót 3-ra állítja.

A megszakítási pontokat a BRKCLR-hez igen hasonló SLOWCLR-rel törölhetjük.

A beállított megszakítási pontokat (csak a software-típusúakat) a

DISPLAY BRK

paranccsal kilistázhatjuk.

Ha a STEP utasítással kapott kiíratás kifutott a képernyőről, a

SHOW <lépésszám>

utasítással az utoljára végrehajtott, legfeljebb 128 utasításnak megfelelő outputot újra kinyomtathatjuk. A SHOW csak a STEP-pel nyomozott utasításokra vonatkozik, teljes sebességgel végrehajtott kódot utólag nem lehet nyomozni vele. A SHOW egyszerre csak egy képernyőnyi outputot ír ki, RETURN megnyomásával tovább lapozhatunk.

A lépésenkénti (STEP vagy SUB utasítással kiváltott) végrehajtás során lehetőség van arra, hogy a memória bizonyos részeit figyelemmel kísérjük. Erre szolgál a

WATCH <név><cím>

parancs, ahol a <név> tetszőleges mnemonikus változó, amelynek kezdőértéke a <cím> lesz. A debugger egy külön ablakot nyit a képernyő tetején, mégpedig minden WATCH utasítás hatására egy sort. A sorban a <cím>-től számított 8 byte értéke jelenik meg DUMP formátumban és úgy változik, ahogy a memória tartalma, a STEP vagy SUB futása során. Legfeljebb 19 WATCH változó adható meg. A változók a későbbiek során

egyszerű értékadással új értéket nyerhetnek, ekkor az ablakban a terület tartalma jelenik meg:

```
30  WATCH  ABLAK1  $800
40  WATCH  ABLAK2  $880
50  STEP 30 PC $900 DISP
60  ABLAK1=$830
70  STEP 30
```

A fenti programrészlet egy, a \$900-as címen kezdődő program nyomkövetését vezérli. A 30-as és 40-es sorban kétsornyi ablakot veszünk el a debugger képernyő-területéből annak felső részén. A legfelső sorban a \$800-\$808-ig, az alatta lévőben a \$880-\$888-ig terjedő memoriabyte-ok tartalmának változását kísérhetjük figyelemmel. Az 50-es utasítás végrehajtása – 30 utasítás lépésenkénti nyomkövetése – közben a képernyő alsó 22 sorában a nyomkövetési információk futnak, a felső két sor helybenmarad. A 60-as utasítás átállítja a legfelső sorban látható memória-ablakot a \$830-\$838-ig terjedő területre. Ez lesz látható a 70-es utasítás hatására a nyomozott 30 gépikódu utasítás végrehajtása során.

A nyomozás során is, de különösen az igényeknek eleget tevő, helyesen működő program vizsgálatánál nyújt nagy segítséget **a ciklusszámlálók használatának lehetősége**. A ciklusszámlálót a

TIMESET <név>

vagy a

TIMESET <név1><név2>

utasítással aktivizálhatjuk és egyidejűleg nullázhatjuk. A második alakot akkor alkalmazzuk, ha előreláthatólag a ciklusszáma meghaladja a 65535-öt, ekkor a név2 a négybyte-os számláló felső helyiértékű byte-jait tartalmazza. A számláló (vagy az egyidejűleg érvényben lévő számlálók) minden gépi ciklusnál eggyel, ezenkívül eggyel laphatáron történő átlépésnél és minden ugró utasításnál. A számlálás csak a STEP vagy SUB idejére vonatkozik. A számlálók értéke a

DISPLAY CYCLE

utasítással írhatjuk ki. Leállításuk a

TIMEHALT <név>

utasítással történik. A ciklusszámlálókat igen előnyösen alkalmazhatjuk a kód hangolása során, annak felderítésére, hogy a program melyik részén tartózkodik legtöbbet a vezérlés, hiszen ennek optimalizálásával jelentősen felgyorsíthatjuk a program egészének lefutását. Ugyancsak jól használható ez a lehetőség a különböző módon, ugyanarra a feladatra megírt programok futási időszükségletének összehasonlítására (benchmark).

A nyomozáshoz tartozó utolsó parancskör a tartományokra vonatkozik. A tartomány a memóriaterület egy olyan része, amelyen elhelyezkedő programokat a debugger minden kérdés nélkül teljes sebességgel hajtja végre, akár STEP, akár SUB van érvényben.

A tartományokban a végrehajtás akkor is teljes sebességű, ha a STEP-et NODISP paraméterrel adtuk ki.

A tartományt egy változó kettős (!) értékadásával és a SUBRANG parancssal definiálhatjuk:

```
<név> = <kezdőcim>  
<név>. = <végcim>  
SUBRANG <név>
```

A formátum önmagáért beszél. Figyeljünk fel a második - végcimet megadó - utasításban szereplő pontra!

A tartományokat a

```
SUCLR <név>  
vagy a  
SUCLR ALL
```

utasítással törölhetjük. Utóbbi valamennyi tartományt törli.

A tartományokat, lassu megszakítási pontokat és ciklusszámlálókat a debugger együtt kezeli. Belőlük összesen egyidejűleg 32 lehet érvényben. A már említett

```
DISPLAY CYCLE
```

utasítás valamennyit kiírja.

Befejezésül egy tartomány-példa. A

```
KERNAL = $E000  
KERNAL. = $FFFF  
SUBRANG KERNAL
```

utasítássorozat végrehajtása után a kernal rutinokat minden esetben teljes sebességgel hajtja végre a debugger.

#### 4.5.8 Csak program-módban használható utasítások

Az első ilyen utasítás az

```
IF <feltétel> THEN <utasítások> ELSE <utasítások>
```

struktúra.



A feltétel és az utasítások, valamint az utasítás-struktúra jelentése a szokásos, így felesleges részletezni. Az utasításokat kettősponttal választjuk el az IF mindkét ágában. A feltételekben használható operátorok:

- > - nagyobb
- < - kisebb
- >= - nagyobb vagy egyenlő
- <= - kisebb vagy egyenlő
- <> - nem egyenlő
- # - nem egyenlő
- = - egyenlő
- AND - és
- OR - vagy.

Megengedett a karakteres változók és konstansok, valamint numerikus kifejezések összehasonlítása. Ekkor a numerikus kifejezés kiértékelésre kerül és az általa hivatkozott memória-területen lévő adatok kerülnek a karakteres kifejezéssel összehasonlításra. Vigyázat, az összehasonlítás karakteres esetben a rövidebb string hosszáig tart! Így 'ABCD' egyenlő lesz 'ABC'-vel! Az AND és OR műveleti jelek zárójelek nélkül nem írhatók.

A debugger programban bárhol elhelyezhetők **cimkék**. A címke egy utasítás helyén álló azonosító, amely csak egyszer fordulhat így elő a programban, egyébként csak GOTO utasítás után. A címke állhat egyedül egy sorban, vagy bárhol kettőspont után, sor közben vagy sor végén is.

A címkére (pontosabban az öt követő utasításra) GOTO utasítással adhatjuk a vezérlést. A GOTO IF THEN vagy ELSE ágában írva feltételtől függő vezérlésátadást tesz lehetővé, ami különösen érdekes lehet STEP utasítás után, annak megvizsgálására, hogy a STEP hogyan zajlott le. Ez a **BRKFLG** előre definiált változó alkalmazásával lehetséges, aminek értéke

- 0 - ha a STEP utasítás rendesen befejeződött,
- 2 - ha lassu megszakítási pont miatt,
- 3 - ha software-megszakítási pont miatt maradt abba,
- >3 - ha bármely billentyű megnyomására szakadt meg, ilyenkor a BRKFLG a billentyűnek megfelelő karakter ASCII kódját tartalmazza.

A fent elmondottakat az alábbi példa segítségével világítjuk meg:

```
10 PLUSZ = 43
20 MINUSZ = 45
30 SBETU = 83
40 LOAD "PROGRAM" $C000
50 BRK MP $C100
60 ? "A PROGRAMOT LEPESENKENT HAJTOM VEGRE"
```

```

70 ? "MEGSZAKITAS S-SEL"
80 ? "AZ ABLAKOT +-SZAL 8 BYTE-TAL FELJEBB"
90 ? " --SZAL 8 BYTE-TAL LEJJEBB"
100 ? "TOLHATOD"
110 WATCH' ABLAK $C200
120 PC = $C000
130 UJRA:STEP 0 DISP
140 IF BRKFLG=0 OR BRKFLG=3 OR BRKFLG=SBETU THEN GOTO VEGE
150 IF BRKFLG=PLUSZ THEN ABLAK=ABLA+8:GOTO UJRA
160 IF BRKFLG=MINUSZ THEN ABLAK=ABLA-8:GOTO UJRA
170 GOTO UJRA
180 VEGE:?"VEGE A TESZTNEK!"

```

A program lépésről lépésre:

- 10-30 - az ASCII kódok beírása a megfelelő változókba
- 40 - a "PROGRAM" nevű gépiódu program betöltése lemezről a \$C000 címtől kezdődően
- 50 - software megszakítási pont beállítása \$C100-ra
- 60-100 - kezelési utasítás kiírása
- 110 - ablak definiálása a képernyő legfelső sorába, a \$C200-\$C208-ig terjedő memória-területre
- 120 - az utasítássláló beállítása \$C000-ra, a program kezdőpontjára
- 130 - a lépésenkénti végrehajtás megindítása. Az utasításslám 65535, minden lépés után kijelzést kérünk. A nyomozást valószínűleg nem az utasításslám-korlát, hanem a megszakítási pont fogja leállítani
- 140 - a STEP befejeződésének vizsgálata. Ha
  - az utasításslám lejárása,
  - a \$C100-on való megszakítás vagy
  - az S-betű megnyomása miatt
 következett be, a VEGE címére ugrunk
- 150 - ha a megszakítás oka a + megnyomása, az ablakot 8 byte-tal feljebb állítjuk és folytatjuk a nyomozást
- 160 - mint a 150, de - billentyűre és az ablak memória-területének csökkentésére
- 170 - minden egyéb megszakítást véletlen billentyű-megnyomásnak tekintünk és folytatjuk a nyomozást
- 180 - program vége, bucsüzenet.

#### 4.5.9 Kód-foltozás

Ha a bejáratás alatt lévő programban hibát találunk, több lehetőség közül választhatunk. Az első és legegyszerűbbnek tűnik az, hogy **helyben kijavítjuk a kódot**, pl. a beépített assembler segítségével. De ez nem olyan problémamentes, mint az első pillanatban látszik. Ha a javítás rövidebb, NOP utasításokkal tolhatjuk fel a felszabaduló programterületet, ha viszont hosszabb, ez az eljárás nem alkalmazható. Ilyenkor látszólag nem marad más hátra, mint **forrásszinten kijavítani a programot**, majd újra lefordítani. Ez persze elég sokáig tart. A harmadik megoldás, amiről itt szó lesz, a **kód-foltozás**. Ekkor a javítás helyén egy ugro-utasítást helyezünk el a memória egy használaton kívüli részére, majd ott beírjuk a szükséges kiegészítést és visszaugrunk a programba. Persze közben igen könnyű hibázni és az ilyen hevesített javítások által bevitt hibák felderítése általában a legkellemetlenebb feladat.

Ezért is nagyon kellemes szolgáltatása a PTD debuggernek, a kód-foltozás támogatása, ezáltal hibalehetőségeinek kiküszöbölése. A foltozás első lépése annak a pontnak a kijelölése, ahol a programba az új kódot illesztjük. Ezt a

PATCH <név> <programcim> <új cim>

utasítással tehetjük meg. A <név> a folthoz rendelt név, <programcim> ennek a pontnak a programbeli címe, <új cim> az új kód elhelyezésére kijelölt terület kezdőcíme. Az utasítás hatására a <programcim>-en lévő byte-ot a debugger kiemeli és helyére egy BRK utasítást tesz, éppen úgy, mint a software megszakítás esetén. A kimentett utasítást a DUP utasítással vihetjük át az új területre, ezáltal elérhetjük, hogy a kód csorbitatlan maradjon. Az új területre a kódkiegészítést a rezidens assemblerrel fordíthatjuk le. A kódrészlet utolsó utasítása egy

RET <név>

alakú utasítás, amely a kodba egy JMP utasítást helyez el vissza az eredeti programba, a foltozási pontot követő utasítására. A kicsit bonyolult eljárást legjobban egy egyszerű példával világíthatjuk meg.

Tegyük fel, hogy a következő programot kívánjuk bejáratni:

```
10 ; MEGNOVELJUK AZ AKKUMULATOR TARTALMAT
20 ; $FE-VEL ES AZ EREDMENYT KETBYTE-OSAN
30 ; AZ X-Y REGISZTEREKBE TESSZUK
40 ASM $B00
50 CLC
60 ADC #$FE
70 TAY ; ALSO BYTE Y-BA
```

```

80 LDA #0
90 ROL A ; ATVITEL A-BA
100 TAX ; FELSO BYTE X-BE
110 RTS
120 RET

```

A programot RUN-nal lefordíthatjuk, ezáltal a kód a \$800-ra került. A szubrutin kipróbálása közben jövünk rá, hogy az azt felhívó program nem is X-Y-ban, hanem a \$880-\$881 címen várja az eredményt. A javítást foltozással oldjuk meg. A program visszafordításával:

```
DISASM $800 10
```

megtudhatjuk, hogy a TAY utasítás címe \$803, a TAX-é \$807. Ezek lesznek a foltozási pontok:

```

PATCH YT $803 $900 ; ELSO FOLT. PONT
PARAM=$880 ; PARAMETER
ASM $900 ; AZ ELSO FOLT FORDITASA
DUP YT ; ATMASOLJA A FOLT. PONTOT
STA PARAM ; MENTES, ALSO BYTE
RET YT ; FOLT VEGE, FORDITAS VEGE
PATCH XT $807 $910 ; MASODIK
ASM $910 ; MASODIK FORDITAS
DUP XT ; MASOLAS
STA PARAM+1 ; MENTES, FELSO BYTE
RET XT ; FORDITAS VEGE
STEP 11 PC $800 DISP ; TESZT

```

A foltozás egyetlen hibája, hogy a programot így nem tudjuk kimenteni, a forrásszövegen újra el kell végeznünk a javítást.

A debugger együtt kezeli a BRK és PATCH utasítással definiált pontokat. Megnyilvánul ez abban is, hogy azt a

```
DISPLAY BRK
```

utasítással listázhatjuk ki és a BRKCLR utasítással törölhetjük. Bánjunk óvatosan a

```
BRKCLR ALL
```

utasítással, mivel az összes foltozást is tönkreteszi!

Ha a programot teljes sebességgel kívánjuk kipróbálni, ezt az

```
EXECUTE <cím>
```

paranccsal indíthatjuk el. Ilyenkor a foltozások is működnek, mégpedig szintén teljes sebességgel. A <cím> elhagyása esetén a program az utasításszámláló (PC) értékétől indul. A programban előforduló BRK utasítások segítségével ill. a RESTORE gomb megnyomásával visszakerülhetünk a debuggerbe.

#### 4.5.10 A PTD debugger értékelése

A fentiekben egy olyan programozási eszközt ismertünk meg, amely - ha némely megoldásán vitatkozni is lehet, mégis - egy példaszerűen megoldott program-bejáratási környezetet ad a 6510-es assembly programok fejlesztésében. Kiválóan alkalmas arra, hogy a program tesztelését a programozással párhuzamosan megtervezzük, a programban vizsgálati pontokat írjunk elő, ott bizonyos feltételek teljesülését várjuk el, programozott teszt-ágyat készítsünk. Alkalmas a szubrutinok egyenkénti kipróbálására, ennek során a környezet szimulálására. Segítségével az elkészült programot elemezhetjük, hangolhatjuk, sebességi összehasonlító vizsgálatokat végezhetünk. Nagyobb lélegzetű assembly programozási munka esetén feltétlenül javasoljuk alkalmazását.

Nem csökkenti nagymértékben a software-termék értékét néhány bántó hiányossága sem, amelyek legfontosabbjai

- a szintaxis nem következetes,
- a monitor-szolgáltatások (memóriamódosítás, keresés) szegényesek,
- az INPUT utasítás rettenetesen hiányzik.

A monitor-szolgáltatások szinte teljes hiányát nyilván az okozza, hogy a debugger eredetileg APPLE mikrogépre készült, ahol a monitor ROM-ból elérhető, de ez a Commodore-n szerény vigasz. Pótolható ez egy egyszerű monitor betöltésével, ha van erre helyünk a tárban. A többi problémára nem tudunk orvosságot.

## 5. Mit programozunk, hogy programozzuk assembly nyelven?

Az eddigi fejezetek az **assembly programozás eszköztárát** igyekeztek ismertetni. Ennek a fejezetnek a célja az **assembly programozás módszertanának bemutatása**. Ez sokkal nehezebb feladat, mint az 1-4. fejezet lexikális jellegű összefoglalásai.

A fő nehézséget az okozza, hogy egyaránt kívánunk szolgálni az érdeklődő amatőr és a tanuló (majdani) profi igényeinek, sok érdekes részletét fel kívánjuk villantani a programozás eme fejezetének, de meg is kívánjuk alapozni a következetes munkát.

Már az első fejezetben volt róla szó, de nem lehet eléggé kihangsúlyozni azt, hogy maga a programozás a feladat megoldásának fontos, de nem egyetlen, sőt nem is a legelső lépése. Ha azt a - címben feltett - kérdést kívánjuk megválaszolni, hogy mit programozunk assembly nyelven, először is le kell szögezni, hogy a programozási nyelv - és az egyéb felhasználandó software-eszközök - kiválasztásának kérdése csak a probléma pontos felmérése és az igények megismerése után dönthető el. Általánosságban kimondható, hogy csak azt programozzuk assembly nyelven, ami elkerülhetetlen. Elkerülhetetlen azért, mert a kívánt hatékonyság, speciális lehetőségek igénybevétele, stb. csak ezzel az eszközzel valósíthatók meg. Ne programozzuk assembly nyelven azokat a feladatokat, vagy feladatrészeket, amelyek magasszintű nyelven is elfogadhatóan megoldhatók, mert az assembly programozás emberi és gépi időráfordítása sokszorosa a magasabb szintű feladatmegoldásának. Ne habozzunk, ha egy feladatot részekre kell szabdalni és egyes részeit assembly, más részeit magasabb szintű nyelven (pl. BASIC) kell megoldani. A többletráfordítás, amit a különböző nyelvű részek interface-inek kidolgozására fordítunk, busásan megtérül a magasszintű nyelven való programfejlesztés hatékonyságával.

A fenti mondatban előfordult egy szó - **interface** - amelyet sok értelemben, sokféle célra használunk. Akár hardware, akár software értelemben beszélünk róla, mindig részrendszerek kapcsolatának jellemzésére szolgál. Ha kicsit képszerűen akarjuk lefordítani az angol eredetit, azt írja le, hogy az egyik részrendszer milyen arcát fordítja a másik felé. A programozás során lépten-nyomon programegységekkel és azok interface-ével találkozunk, amelyek pontos, hatékony megfogalmazása a feladat dekomponálását jelenti és óriási lépés a megoldás felé. Különösen fontos az interface tökéletes ismerete, ha a programot egy más által írt programrészlethez kívánjuk illeszteni, amelyre bőven találunk példát ebben a fejezetben. Ebben az esetben nem az interface meghatározása - mivel abba már nincs beleszólásunk, hanem alapos megismerése a záloga a sikeres feladatmegoldásnak.

## 5.1 Memóriaigazdálkodás

A Commodore 64 memóriafelosztásának ismerete inkább a mikroprocesszor ismertetéséhez kívánczogna, itt viszont nem hardware-leírást akarunk róla adni, inkább azt a nagyon is gyakorlati kérdést próbáljuk megválaszolni, hogy **hová helyezze el az assembly programozó programját** és adatait. Különösen fontos ez abban az esetben, ha nem önálló programot írunk, hanem – mint általában – más által írt software-hez, leggyakrabban a BASIC rendszerhez, a KERNAL-hoz és esetleg egy felhasználói BASIC programhoz is akarjuk kapcsolni programunkat. Ekkor nagyon fontos, hogy ne használjunk olyan memóriaterületet, amelyet az említett, együttműködő software-ek is használnak, olyan célra, amely azok funkciójával összeférhetetlen. Ezért ebben a fejezetben memóriaigazdálkodáson mindazokat az ismereteket értjük, amely a hardware, a fontosabb firmware-ek és software-ek memóriahasználatának jellemzése.

A Commodore 64 memóriája 8 Kbyte-os részekre van osztva, amelyek közül néhányan több példányban is létezhetnek. Ez azt jelenti, hogy fizikailag ugyanazon a memóriaterületen több memóriabyte (általában kettő) is található. Pontosabban, azokon a címeken (\$A000-\$BFFF, \$D000-\$DFFF, \$E000-\$FFFF), ahol ROM van, RAM is van, némiképpen elrejtve. Ha az ilyen kettős memóriabyte-okat írjuk, az érték mindig az adott című RAM-ba. Ha viszont olvassuk, akkor vagy a ROM vagy a RAM tartalmát látjuk, attól függően, hogy az 1. című input-output regiszter megfelelő bitjeinek értéke milyen. A bitek és a memóriaszeletek összefüggése a következő:

|        |               |
|--------|---------------|
| 0. bit | \$A000-\$BFFF |
| 1. bit | \$E000-\$FFFF |
| 2. bit | \$D000-\$DFFF |

Ha a megfelelő bit értéke magas, a mikroprocesszor az adott címen lévő ROM-byte-ot olvassa, ha alacsony, a RAM-byte-ot. A \$D000-\$DFFF-ig terjedő terület kezelése kicsit speciális, mivel annak egyes részein három memória-chip fedi át egymást. A ROM-on és a RAM-on kívül erre a területre esnek a négy interface-chip, a VIC II képernyőkezelő, a SID hangvezérlő és a két CIA általános input/output vezérlő chip regiszterei. Ezek képesek arra –, pontosabban a VIC II képes arra –, hogy amikor szükséges, olvasni tudjon a \$D000-tól a \$DFFF-ig terjedő ROM-ból, amely a képernyőn megjeleníthető karakterek képét tartalmazza.

**A ROM-ok alatti RAM-ok használatára** rövidesen visszatérünk, de előtte szóljunk egy pár szót a memória egyéb részeiről. Nem akarjuk itt részletesen ismertetni a memória fontosabb címeit, hiszen azt igen sok munkában megtalálhatja az olvasó (ld. a mellékelt irodalom-jegyzéket), hanem inkább a

memória kevésbé fontos részeit akarjuk feltérképezni, azokat a területeket, ahova elhelyezhető általunk írt program.

Ha egymagában működő gépikódu programot akarunk készíteni - ami elég ritka - az 1. byte megfelelő állításával rendelkezésünkre áll a teljes 64 K memória. Ekkor is kerülendő a 0-ás és 1-s lapon a programok tárolása, hiszen - különösen az utóbbit - a processzor speciálisan használja.

Megjegyzendő, hogy az I/O chipek a \$D000-\$DFFF-ig terjedő memóriaterület egyes részei csak olvashatók. 64K RAM (igazi) csak a bővítő egység (cartidge) GAME és EXROM vonalainak aktivizálásával érhető el.

Ha programunkban támaszkodunk a KERNAL rutinokra és/vagy a BASIC ROM-ra, az alattuk lévő RAM-ot már csak részlegesen és csak abban az időpillanatban használhatjuk, amikor a fenti ROM-okra nincs szükség.

Ha BASIC programmal is együtt kívánunk működni, a helyzet bonyolódik. Ilyenkor is szabadon használhatók:

- a nullás lapon az \$FB-\$FF címek,
- ha nem használunk kazettás egységet, az \$A9-\$AB-ig terjedő címek is,
- átmenetileg egy sereg byte a nullás lapon, példaként \$4E-\$60 és \$26-\$29 vagy \$14-\$15, amelyeket egy SYS BASIC utasítással hívott assembly rutinban elronthatunk, de a BASIC is írhatja azokat,
- \$200 feletti címen a \$334-től \$3FF-ig (ha nem használunk kazettás egységet),
- a \$C000-től \$CFFF-ig.

A fentieken kívül, amelyek közül a \$334-\$3FF rövidebb BASIC kiegészítések, a \$C000-\$CFFF hosszabb programok számára előnyös, ellophatunk helyet a BASIC-programok tárolására szolgáló területből is. A BASIC-programok tárolására rendelt memóriarész (\$0B01-\$9FFF) felosztását a következő pointerok mutatják:

- |             |   |                                                                                                                                  |
|-------------|---|----------------------------------------------------------------------------------------------------------------------------------|
| \$2B - \$2C | - | BASIC program kezdete (rendszerint \$B01)                                                                                        |
| \$2D - \$2E | - | BASIC változók kezdete, amely általában közvetlenül a program után következik, értéke annak hosszától függ                       |
| \$2F - \$30 | - | BASIC tömbök kezdete                                                                                                             |
| \$31 - \$32 | - | BASIC tömbök vége+1                                                                                                              |
| \$33 - \$34 | - | stringek kezdete+1 (vigyázat, ez a legmagasabb cím, amin string van, innen visszafelé terjed a \$31-\$32-ig), rendszerint \$A000 |
| \$37 - \$38 | - | BASIC terület vége, rendszerint \$A000                                                                                           |

Ezen pointerok állításával helyet szabadithatunk fel assembly programunknak. Így pl. a \$2B-\$2C feljebb állításával a \$B01 és a beállított cím közötti terület teljesen szabadon használható. A BASIC memória végén a \$33-\$34 és a \$37-\$38 lejjebb állításával nyerhetünk memóriát \$A000-ig, ahol a BASIC ROM kezdődik. Egy betöltött BASIC program mögött a



\$2D-\$2E feljebb állításával szabadithatunk fel memóriaterületet a régi értéktől (a program valóságos vége) az új értékig (a BASIC változók kezdete). Mindezen műveletek után ki kell adni egy CLR BASIC utasítást vagy egy JSR \$A56E assembly utasítást, hogy a változtatásokat a BASIC interpreter tudomására hozzuk. Így gyakori megoldás, hogy az assembly betétek a BASIC program mögött vannak, és azzal együtt töltődnek be a tárba. Ekkor a SAVE az egészet kimentti. Természetesen a BASIC program ilyenkor már nem szerkeszthető!

Térjünk vissza a ROM alatt fekvő RAM használatához. Ha ezt úgy akarjuk megtenni, hogy egy BASIC programmal együttműködjünk vagy a KERNAL-t használjuk, körülményes ide programokat tölteni. Az ilyen programrészlet futása alatt a ROM inaktív, tehát ahhoz nem fordulhat, még hívását is egy, a normál RAM-ban lévő címről kell megvalósítani az 1. byte megfelelő átállítása után. Kiválóan alkalmas viszont ez a terület adattárolásra. Érdemes ebből kizárni a \$D000-\$DFFF-ig terjedő részletet, amelynek hozzáférése körülményesebb, így is 16 Kbyte tárolóterületet nyerünk, ami nem is kevés a BASIC-terület mintegy 38 K-jához viszonyítva. A következő program ezt a területet teszi elérhetővé, mégpedig BASIC programból. Ezáltal egy kicsit előreszaladunk, belekóstolunk a BASIC-kiegészítések világába, amiről még lesz szó.

A program feladata a BASIC terület végétől a tár végeig terjedő RAM használata, eltekintve a \$D000 és \$DFFF közé eső címektől, a határokat is beleértve. Ezt a területet, mint egyetlen összefüggő mezőt kezeli a program, sorszámozása 1-től addig a végcímig terjed, amelyet a PIK belépési pont 0 címmel történő hívása ad vissza (USR(0)). A programnak három belépési pontja van. A POK nevű (SYS828) a POKE BASIC utasításhoz hasonló, a két elvárt paraméter a memóriaterületen belüli (relatív) cím és a byte-on elhelyezkedő érték. A RESET nevű belépési pontot (SYS918) célszerű legelőször meghívni a BASIC programból, mivel ez az USR vektort a PIK belépési pontra állítja. Ezután USR(cím) alakú függvényhívással - a PEEK BASIC utasításhoz hasonlóan - a tárolóterület adott című byte-jának értékét kapjuk vissza. Most lássuk a programot utasításról utasításra:

```

100      ;PIK-POK/ASS
110      ;FELSO TAR<BASIC TERULET VEGETOL-CFFF,E000-FFFF - >=
115      ;20K> HASZNALATA BASICBOL
120      ;POK FUNKCIO SYS 828,CIM,ERTEK-KENT HIVHATO. CIM
125      ;LEGKISEBB ERTEKE 1.
130      ;MAXIMALIS ERTEKET PIK 0 CIMMEL TORTENO HIVASA ADJA
135      ;VISSZA, ERTEK 0-255
140      ;PIK FUNKCIO USR(CIM)-KENT HIVHATO
150      ;RESET FUNKCIO SYS 918-KENT HIVHATO
160 Z1    = $14
170 FAC   = $61
180 Z2    = $A9
190 USR    = $311
200      * = $33C
```

```

210 BASVEG = $37
220 CHRGET = $73
230 GETADR = $B7F7
240 YTOFAC = $B3A2
250 FRMEVL = 4AD9E
260 AYTFAC = $B391
270 POK      JSR CHRGET      ; VESSZO
280          JSR FRMEVL      ; CIM FAC-BA
290          JSR GETADR      ; FAC-BOL CIM FORMABAN Z1-BE
300          JSR NOVEL       ; Z2=Z1+BASVEG
310          JSR CHRGET      ; UJABB VESSZO
320          JSR FRMEVL      ; ERTEK FAC-BA
330          JSR GETADR      ; FAC-BOL CIM FORMABAN Z1-BE
340          LDY #0
350          LDA Z1          ; ALSO BYTE-JANAK
360          STA (Z2),Y      ; TAROLASA A MEGADOTT CIMRE
370          RTS            ; VISSZA A BASIC-HEZ
380 NOVEL    CLC ; Z1-HEZ HOZZAADJUK A BASIC VEGPOINTER ERTEKET
390          LDA BASVEG
400          ADC Z1
410          STA Z2
420          LDA BASVEG+1
430          ADC Z1+1
440          STA Z2+1
441          CMP #$D0        ; IF Z2>D000
443          BMI RETNOV
445          CLC
447          ADC #$10        ; Z2=Z2+$1000
449          STA Z2+1
450 RETNOV   RTS            ; VISSZA
460 PIK      JSR GETADR      ; FAC-BOL USR PARAMETEREKENT MEGADOTT
465          ; CIM Z1-BE
470          LDA Z1          ; HA Z1=0, A MAX CIMET KERDEZIK
480          BNE PIK1
490          LDA Z1+1
500          BEQ MAXCIM
510 PIK1     JSR NOVEL       ; Z2=Z1+BASVEG
520          LDY #0
530          LDA #53        ; ADATOK ELOKESZITESE
540          SEI            ; MEGSZAKITAS MASZKOLASA
550          STA 1          ; ROM KIKAPCSOLASA
560          LDA (Z2),Y      ; ERTEK KIOLVASASA
570          TAY            ; Y-BA
580          LDA #55        ; ROM VISSZA
590          STA 1
600          CLI            ; MEGSZAKITAS ENGEDELVEZESE
610 RET      JMP YTOFAC      ; Y FAC-BA ES VISSZA A BASIC-HEZ
620 MAXCIM   SEC            ; $CFFF-BASVEG
630          LDA #$FF
640          SBC BASVEG
650          TAY            ; ALSO BYTE
660          LDA #$CF
670          SBC BASVEG+1
730          CLC            ; +$2000
740          ADC #$20        ; FELSO
750          JMP AYTFAC      ; FAC-BA ES VISSZA A BASIC-HEZ
760 RESET    LDA #PIK<      ; USR -> PIK
770          STA USR
780          LDA #PIK>

```

- 270-290. sor - A POK belépési pontnál az első feladat a paraméterátvétel. A CHRGET hívása átlépi a BASIC szövegben szereplő soronkövetkező karaktert, ami a SYS828 után álló vessző. A FRMEVL interpreter rutin, mint a későbbiekben részletesen is tárgyaljuk majd, az első paraméter (a cím) értékét tölti lebegőpontos formában a FAC lebegőpontos akkumulátorba, amelyet a következő GETADR hívás konvertál kétbyte-os címmé és helyez Z1-be (\$14-\$15).
300. sor - A NOVEL szubrutin, amelyet alább még megvizsgálunk, ezt az értéket megnöveli a BASIC terület végmutatójával, ami a \$37-\$38 címen található és értéke alaphelyzetben \$A000.
- 310-330. sor - Itt az értéket - második paraméter - helyezzük az előbbieken megismert módon Z1-be.
- 340-360. sor - Z1 tartalmát lerakjuk a kiszámított címre.
370. sor - POK szubrutin vége, visszatérünk a hívó BASIC programba.
- 380-450. sor - A már említett NOVEL szubrutin Z1+BASVEGET helyezi el Z2-ben.
460. sor - Az USR-rel hívott PIK belépési pont. Az USR hívása esetén az interpreter már előre gondoskodik a zárójelben lévő érték FAC-ba töltéséről, így csak a második lépés, a GETADR-ral történő konvertálás marad.
- 470-500. sor - Megvizsgáljuk Z1 értéket. Ha az kétbyte-os nulla, a megállapodás szerint a legnagyobb megadható címet kérdezik, MAXCIM-re ugrunk.
510. sor - A már ismert NOVEL hívása, az abszolút cím kiszámítása BASVEG hozzáadásával.
- 520-550. sor - Itt történik a ROM-ok kikapcsolása. Az akkumulátorba betöltjük az 1. byte szükséges értékét, maszkoljuk a megszakításokat és átállítjuk az 1. című byte-ot. A megszakítás maszkolására azért van szükség - tulajdonképpen ez csak a KERNAL-ROM-ra vonatkozik -, mert egy érkező megszakítás a kikapcsolt részre adná a vezérlést, ahol most nem a megszakítás-kezelő szubrutin, hanem a mi adataink vannak.

- 560-570. sor - Az adatbyte kiolvasása és áttöltése az Y regiszterbe.
- 580-600. sor - Visszakapcsoljuk a ROM-ot és engedélyezzük a megszakítást.
610. sor - Ez a szubrutin utolsó utasítása, de ez elég sok magarázatot kíván. Mivel a kiolvasott értéket a FAC-ba kell helyeznünk a visszatérés előtt, itt is a megfelelő interpreter rutint hívjuk meg a feladat elvégzésére, ahelyett, hogy megírnánk újra. Ez a \$B3A2-n lévő YTOFAC-nak nevezett szubrutin, amely az Y regiszter értékét lebegőpontosra konvertálva a FAC-ba tölti. Erdemes figyelni egy, az assembly programozásban szokásos fogásra. A szubrutint, amelynek hívása a mi szubrutinunk utolsó utasítása, nem JSR-rel hívtuk meg, hanem JMP-pal ug-runk rá. Az interpreter-rutin végén lévő RTS így nem a mi programunkra, hanem egyenesen a minket hívó programhoz tér vissza, hiszen annak visszatérési címe van a verem tetején. Ez az eljárás, amely nem nagyon strukturált, és hatása a programból csak némi gyakorlattal olvasható ki, igen hatékony, mivel megtakarítunk vele egy címelmentési-visszamentési műveletet és 2 byte-ot a veremben, ami néha nem elhanyagolható szempont.
- 620-740. sor - A már hivatkozott MAXCIM programrész feladata a megengedett maximális cím visszaadása. Ezt a \$CFFF-BASVEG+\$2000 kifejezés adja meg. A \$CFFF-BASVEG a \$D000-ig, a \$2000 az azon felül rendelkezésre álló byte-ok száma.
750. sor - A kiszámított értéket, amelyet Y-ba (alsó helyiértékű byte) és A-ba (felső helyiértékű byte) helyeztünk, az AYTFAC interpreter rutin az előzőekben használt YTOFAC-hoz hasonlóan - de két byte-ból összerakva - helyezi el a FAC-ban, ami az USR függvény visszatérési értékét kell, hogy tartalmazza a szubrutin lefutása után.
- 760-800. sor - A RESET szubrutin feladata csupán az, hogy PIK címét az USR vektorba (\$311) töltsse.

A következőkben ugyanezt a területet egy kicsit bonyolultabban, de sok esetben praktikusabban használjuk fel. A sortároló rutin úgy működik, mint egy FIFO-verem, azaz egymásután tölthetjük bele a byte-okat, majd a belehelyezés sorrendjében ki is szedhetjük őket anélkül, hogy a soronkövetkező byte címét tudnunk kéne, azt a szubrutin maga tárolja. Először nézzük magát a programot: 86

```

110      ; MC RUTIN BASIC SORTAROLOHOZ. TAROLAS BGADR-$CFFF,
115      ; $E000-$FFFF (>=20K)
120      ; PUSH FUNKCIO USR(ERTEK)-KENT HIVHATO, 0 HA OK, -1
125      ; HA NEM (TUL SOK PUSH)
130      ; PULL FUNKCIO USR(0)-KENT HIVHATO, ERTEK HA OK, -1
135      ; HA NEM (TUL SOK PULL)
140      ; SETIN FUNKCIO SYS945,BGADR-KENT HIVHATO, ADATIRANY:
145      ; PULL
150      ; SETOUT FUNKCIO SYS977-KENT HIVHATO, ADATIRANY: PUSH
160 Z1      = $14
170 FAC      = $61
180 Z2      = $A9
190 USR      = $311
200          = $33C
205 CHRGET   = $73
210 GETADR   = $B7F7
220 YTOFAC   = $B3A2
230 FRMEVL   = $AD9E
240 ILEGAL   = $B248
250 PUSH      JSR ADRINC      ; KOVETKEZO CIM
270          JSR GETADR      ; USR-BEN MEGADOTT ERTEK FAC-BOL
272          ; Z1-BE
275          LDY #0
280          LDA Z1
290          STA (Z2),Y      ; TAROLASA A KOVETKEZO CIMRE
300          LDA #0          ; FAC=0 -> OK
302          STA FAC
304          STA FAC+1
306          JMP RETOK      ; VISSZA A BASIC-HEZ
310 PULL      LDA NXTADR      ; VEGE A BEIRT ERTEKEKNEK PRINT
320          CMP ENDADR
330          BNE ADR3
340          LDA NXTADR+1
350          CMP ENDADR+1
360          BEQ OVER1      ; IGEN -> HIBAJELZESSEL VISSZA
370 ADR3      JSR ADRINC      ; CIMET NOVEL
380          LDY #0
390          LDA #53        ; ADATOK ELOKESZITESE
400          SEI            ; MEGSZAKITAS MASZKOLASA
410          STA 1          ; ROM KIKAPCSOLASA
420          LDA (Z2),Y      ; ERTEK KIOLVASASA
430          TAY            ; Y-BA
440          LDA #55        ; ROM VISSZA
450          STA 1
460          CLI            ; MEGSZAKITAS ENGEDELYEZESE
470 RET      JMP YTOFAC      ; Y FAC-BA ES VISSZA A BASIC-HEZ
480 ADRINC    INC NXTADR      ; CIMET NOVEL
490          BNE ADR1
500          LDA NXTADR+1
510          CLC
520          ADC #1
530          BEQ OVERFL      ; CIM=0000 -> MEMORIA BETELT
540          CMP #$D0        ; HA A CIM >=$D000 ES , $E000
550          BCC ADR2
560          CMP #$E0

```

```

570      BCS ADR2
580      CLC
590      ADC ##10      ; AKKOR HOZZAADUNK $1000-T
600 ADR2  STA NXTADR+1  ; UJ CIM FELSO BYTE
610 ADR1  LDA NXTADR    ; UJ CIMET Z2-BE
620      STA Z2
630      LDA NXTADR+1
640      STA Z2+1
650      RTS
660 OVERFL PLA          ; VISSZATERESI CIM VEREMBOL
670      PLA
680 OVER1 LDA ##B1      ; FAC=-1
690      STA FAC
700      LDA ##80
710      STA FAC+1
715      STA FAC+5
720      LDA #0
730 RETOK STA FAC+2      ; VISSZA A BASIC-HEZ
740      STA FAC+3
750      STA FAC+4
760      RTS
770 SETOUT LDA #PUSH<   ; USR -> PUSH
780      STA USR
790      LDA #PUSH>
800      STA USR+1
810      JSR CHRGET      ; TERMINATOR
812      JSR FRMEVL      ; CIM -> FAC
814      JSR GETADR      ; FAC -> Z1 ($14/$15)
816      LDA Z1
818      STA BGADR
820      LDA Z1+1
822      STA BGADR+1
842 SETOU1 LDA BGADR     ; NXTADR=BGADR
844      STA NXTADR
846      LDA BGADR+1
848      STA NXTADR+1
850      RTS
860 SETIN  LDA #PULL<    ; USR -> PULL
870      STA USR
880      LDA #PULL>
890      STA USR+1
900      LDA NXTADR      ; UTOLSO CIM ELMENTESE
910      STA ENDADR
920      LDA NXTADR+1
930      STA ENDADR+1
940      JMP SETOU1      ; ES NXTADR BEALLITASA AZ ELEJERE
950 NXTADR .BYTE 0,0
960 ENDADR .BYTE 0,0
970 BGADR  .BYTE 0,0

```

A programot érdemes részletesebben is megsejmlélni. Négy belépési pontja van BASIC-ből:

1. A programot - amely a \$33C-től indul - először a SYS 945,KC BASIC utasítással hívhatjuk fel. A hívás a SETOUT belépési pontnak felel meg, amely előkészíti az értékek beírását a memóriába s egyuttal tudomására hozza az assembly programnak a rendelkezésére álló terület kezdőcímét (KC),ez egyszerű esetben \$A000 lehet.
2. Ezután tetszés szerint sokszor hívhatjuk a programot az USR(E) függvény aktivizálásával, ahol E az elhelyezni kívánt byte értéke. A függvény értéke 0, ha a tárolás rendben ment, -1, ha átléptük a memória felső határát (\$FFFF).
3. Az értékek kiolvasása a SYS 987 BASIC utasítással készíthető elő.
4. Az egymásután hívott USR(0) függvények visszaadott értéke a soronkövetkező byte értéke, ill. -1, ha túlléptük a beírt byte-ok mennyiségét. A visszaadás FIFO típusu,tehát az elsőnek beírt értéket kapjuk vissza az első olvasáskor.

Vegyük sorra a belépési pontokat:

770-800. sor - A BASIC USR függvénynek egy vektor felel meg, amelynek címe \$311-\$312.Ha aktivizáljuk az USR függvényt a BASIC programban, a zárójelben lévő kifejezés értéke a FAC lebegőpontos akkumulátorba kerül (\$61-\$65, \$66 a FAC előjelét külön is tárolja) és meghívódik a \$311-\$312 vektor által hivatkozott gépiódu program. A program output paraméterét,amely szintén lebegőpontos szám, a FAC-ban adja vissza. Ezek a programsorok a \$311-\$312-be a PUSH címke által hivatkozott címet töltik, így a következő USR hívás oda adja majd a vezérlést.

810-840. sor - A SYS utasítással hívott gépiódu programrészlet futásakor a BASIC interpreter által olvasandó következő karakter (amelyet a \$73 című CHRGET szubrutin ad meg) a gépiódu program címét meghatározó kifejezést követő byte. Jelen esetben ez egy vessző. Ezt a CHRGET hívásával lépjük át, ellenőrzés nélkül.Ezután egy numerikus kifejezést várunk el,amelynek értéke a felhasználandó terület kezdőcíme. A BASIC interpreter \$AD9E címen kezdődő szubrutinja, (amely a FRMEVL (Formula Evaluation) nevet viseli, a következőket hajtja végre: meghatározza a kifejezés értéket és azt elhelyezi a FAC lebegőpontos akkumulátorban. Ez nagyon jó, de nekünk a cím kétbyte-os egész alakra konvertált alakjára van szükségünk. Ezt a konvertálást teljesen felesleges megírunk,mivel éppen ezt

az interpreter \$B7F7 kezdőcímű GETADR szubrutinja, amely a FAC értékét kétbyte-os, cím formátumu (alacsony byte - magas byte) egészszé alakítja és elhelyezi a \$14-\$15 címeken (Z1). Ezt az értéket áttöltjük a saját területen lévő BGADR és NXTADR változókba. Utóbbi majd a tároló utolsó foglalt címét mutatja.

850. sor - Visszatérés a BASIC programba.
250. sor - Ezt a belépési pontot írtuk a \$311-\$312 vektorba, tehát az USR hívásakor ide kerül a vezérlés. Ekkor a zárójelben lévő kifejezés értéke FAC-ban van. Először a tárolás pointerét (NXTADR) növeljük meg egy byte-tal. ezt az ADRINC szubrutin végzi, amely a 480-650 sorokban helyezkedik el.
- 480-520. sor - Itt növeljük a kétbyte-os NXTADR mutatót. A kisebb helyiértékű byte-ot helyben, a nagyobb helyiértékűt (ha szükséges, laphatáron) az akkumulátorban.
530. sor - Ha a felső byte is nulla, kimerítettük a tárat, több értéket nem tudunk elhelyezni, az OVERFL címkére ugrunk.
- 540-590. sor - Ha a cím \$D000 és \$DFFF közé esik, átlépjük ezt a tartományt úgy, hogy a magasabb helyiértékű byte-ot \$10 hozzáadásával \$E0-ra növeljük. Azoknak, akik a vizsgálatot ( $\geq \$D0$  és  $< \$E000$ ) felesleges óvatosságnak tartják, hiszen a \$D0-ra való egyenlőség-vizsgálat is elegendő lenne, ajánlom tanulmányozásra Murphy törvényeit...
- 600-650. sor - A magas helyiértékű byte tárolása, majd az aktuális cím áttöltése Z2-be (\$A9), amely az indirekt címzés módhoz kell, mint nullás lapon lévő pointer. Ez a kétbyte-os mutató most a tárolásra használt byte címét tartalmazza. Ezután visszatér a hívás helyére.
- 660-760. sor - Mielőtt visszaugranánk, nézzük meg gyorsan a tulcsordulás esetét. Először két értéket kiveszünk a veremből, mivel itt ki fogunk hagyni egy szubrutinból való visszatérési (RTS) utasítást és ennek vektora még a stack-ben van. Persze ez nem szép megoldás, nem strukturált, nem illendő, csak hatékony. Ezután lebegőpontos -1-et töltünk a FAC-ba (\$81,\$80,0,0,0) és \$80-at a FAC külön előjelébe, amely jelzi, hogy a szám kisebb, mint 0. A rutint záró RTS a BASIC programba visz vissza.



- 270-280. sor - Mivel a függvény paramétere már a FAC-ban van, ezt a már jól ismert GETADR szubrutinnal visszük \$14-\$15-re (Z1). Ebből csak az alsó byte-tal foglalkozunk - tehát a 256-os osztás maradékát vesszük, ha a kifejezés nagyobb 256-nál -. Az Y regiszterbe nullát töltünk az indirekt indexelt címzés mód módosító értékeként.
290. sor - Ez a lényeg, a byte-ot lerakjuk a Z2 által mutatott helyre.
- 300-306. sor - A visszatéréshez a 0 függvényértéket (\$61-\$66-ig csupa 0) tesszük el és a RETOK címkén át - kihasználva az ott már egyszer leírt utasításokat - visszatérünk a hívó BASIC programhoz.
- 860-940. sor - Ez a belépési pont (SETIN) az olvasás előkészítése. Az első lépés az USR vektorának átállítása programunk PULL címkéjére. A NXTADR most a legutoljára beírt byte címet mutatja, ezt elmentjük ENDADR-be, majd a SETOU1 címkére ugrunk, ahol ismét a EGADR tartalmát - a terület kezdőcímét - töltjük NXTADR-be és visszatérünk a BASIC programhoz.
- 310-360. sor - Az újonnan beállított USR belépési pont. Az első lépés annak vizsgálata, hogy vége van-e a beírt értékeknek -ENDADR alapján-, azaz az előző lépésben visszaadott byte már az utolsó byte volt-e. Ha igen, visszatérés az OVER1 címkén át, amely megegyezik OVERFL hatásával, kivéve a veremmutató kettővel való növelését, hiszen az itt felesleges (sőt, ártalmas) lenne.
- 370-380. sor - Az ADRINC hívásával a már ismert rutinnal növeljük NXTADR értékét. Itt tulajdonképpen az ADRINC-ben lévő vizsgálat, hogy NXTADR meghaladja-e \$FFFF-et, felesleges, hiszen az ENDADR-ral való összehasonlítás előbb jelez, de így nem kellett egy külön szubrutint írni, meg egyébként is (Murphy...). Az Y regiszter nullára állítása itt is az indirekt indexelt címzést szolgálja.
- 390-470. sor - lásd az előző programhoz fűzött megjegyzéseket.

Ezzel kis programunk ismertetésének a vége értünk, amely több szempontból is tanulságos volt. Először is szemléltette az elrejtett ROM-ok használatát. Másodszor jó példáját adta a BASIC programmal együttműködő program interface-ének, felvonultatva a két fontosabb változatot:

- SYS utasítás esetleges paraméter-átadással, és
- USR függvény alkalmazása és paraméter-kezelése.

Harmadszor bemutatta azt, hogyan kell a BASIC-interpreter rutinokat felhasználni az assembly nyelvű programbetétekben a BASIC rendszerrel való kapcsolattartásra.

Ebben a programban több olyan dologról is szó esett, amelyet a későbbiekben részletesebben fogunk tárgyalni, de talán hasznos ezeket az elemeket először egy gyakorlati alkalmazásban látni.

A fenti program egy speciális feladat megoldására szolgál, arra, hogy a BASIC-terület végétől a rendelkezésre álló helyen, amely 20 Kbyte-tól akár 55 Kbyte-ig is terjedhet, hogyan lehet egy puffertárolót létesíteni. Jó programozási gyakorlat ennek átírása kicsit más - hasznos - feladatok megoldására, pl. fentiekhez hasonló rutinokká, de nem egy-byte-os értékek, hanem pl. BASIC-lebegőpontos számok (5 byte) tárolására (segítségül: az FRMEVL-lel átvett kifejezés értékét FAC-ból tölthetjük a memóriába és viszont).

## 5.2 Assembly = szerelés

Kicsit kitekerve az assembly szó jelentését - pontosabban az eredeti jelentését véve alapul - azt mondhatjuk, hogy az assembly programozás nem más, mint alkatrészek összeszerelése. Aki egy kicsit is belekóstolt már az assembly programozásba, biztosan észrevette, hogy általában ugyanazon vagy roppantul hasonló elemeket, néhány utasításból álló programrészleteket kell kombinálni. Nyilván ezért is kézenfekvő - ha lehetőségünk van rá - a makróassemblerek használata.

Ismerkedjünk most meg az assembly programozás néhány egyszerű építőkövével, alkatrészével! Hogy a programrészletek jelentése anélkül is nyilvánvaló legyen, hogy sokat magyaráznánk, baloldalt egy BASIC programrészletet, jobboldalt a megfelelő vagy nagyon hasonló assembly utasítássorozatot adjuk meg.

Ertékadások:

|       |                   |
|-------|-------------------|
| AA=12 | LDA #12<br>STA AA |
| B=C   | LDA B<br>STA C    |

Aritmetika egybyte-os értékekkel:

|          |                                                 |
|----------|-------------------------------------------------|
| SM=SM+20 | LDA SM<br>CLC<br>ADC #20<br>STA SM              |
| SM=SM-13 | LDA SM<br>SEC<br>SBC #13<br>STA SM              |
| Z=Z+F-2  | LDA Z<br>CLC<br>ADC F<br>SEC<br>SBC #2<br>STA Z |
| Z=Z*2    | ASL Z                                           |

Z=Z\*10

```
LDA Z
ASL A      ; *2
ASL A      ; *2 -> *4
ASL A      ; *2 -> *8
CLC
ADC Z
CLC
ADC Z      ; B*Z+Z+Z
STA Z
```

N=N+1

```
INC N
```

N=N-2

```
DEC N
DEC N
```

### Programstrukturák:

FOR N=1 TO 5

...

NEXT N

CFEJ

```
LDX #1
...
...
INX
CPX #5
BNE CFEJ
```

FOR N=0 TO 100 STEP 4

...

NEXT

CFEJ

```
LDX #0
...
...
CLC
TXA
ADC #4
TAX
CPX #100
BNE CFEJ
```

FOR N=S+8 TO K

...

NEXT

CFEJ

```
LDA S
CLC
ADC #8
TAX
...
INX
CPX K
BNE CFEJ
```

Megjegyzés: a ciklus legalább egyszer lefut!

FOR N=10 TO 1 STEP-1

...

NEXT

CFEJ

```
LDX #10
...
...
DEX
BNE CFEJ
```

IF K<>0 THEN GOTO 500

```
LDA K
BNE CIM500
```

IF K+3 < F THEN GOTO 500

LDA K  
CLC  
ADC #3  
CMP F  
BMI CIM500

IF FF >= 0 THEN GOTO 500

LDA FF  
BPL CIM500

IF K > 0 THEN K=13

LDA K  
BMI NEXT  
BEQ NEXT  
LDA #13  
STA K

NEXT ...

IF N=0 THEN S=-S

LDA N  
BNE NEXT  
LDA S  
EOR #\$FF ; LOGIKAI KOMPLEMENT  
CLC  
ADC #1  
STA S ; KETTES KOMPLEMENT

NEXT ...

Megjegyzés: S egybyte-os kettes komplement ábrázolású szám.

IF K < 20 AND K > 0  
THEN GOTO 100

LDA K  
CMP #20  
BPL NEXT  
CMP #0  
BPL CIM100

NEXT ...

IF K < F OR K = 32  
THEN GOTO 100

LDA K  
CMP F  
BMI CIM100  
CMP #32  
BEQ CIM100

Kétbyte-os aritmetika:

N=N+1

INC N  
BNE FOLYT  
INC N+1

FOLYT ...

Megjegyzés: a felső byte növelése akkor szükséges, ha az alsó byte 0-ra ugrik.

N=N-1

BIT N  
BNE FOLYT  
DEC N

FOLYT DEC N+1

Az alsó byte-ot nem DEC, hanem SBC utasítással csökkentettük, mivel a DEC nem állítja a C jelzöt, amelyet a 0-ról csökkenés figyelésére használhatunk. A Z jelző nem jó, mert nem abban a lépésben kell a felső byte-ot csökkenteni, amikor az alsó byte 0, hanem a következőben. A programrészletet még sokféleképpen meg lehet írni, ez is egy jó megoldás.

NN=C+B (C és B egybyte-osak,  
NN kétbyte-os)

```
LDA #0
STA NN+1
CLC
LDA C
ADC B
STA NN
BCC FOLYT
INC NN+1
FOLYT ...
```

Megjegyzés: az alsó byte-ba az egybyte-os összeg, a felsőbe a C jelző értéke kerül.

NN=KK+B (NN és KK kétbyte-osak,  
B egybyte-os)

```
LDA KK+1
STA NN+1
CLC
LDA KK
ADC B
STA NN
BCC FOLYT
INC NN+1
FOLYT ...
```

NN=KK+LL (mind kétbyte-os)

```
CLC
LDA KK
ADC LL
STA NN
LDA KK+1
ADC LL+1
STA NN+1
```

Megjegyzés: ez a legegyszerűbb, hiszen a C jelző és az ADC utasítás éppen a kétbyte-os összeadást segíti.

NN=KK-B (NN és KK kétbyte-os,  
B egybyte-os)

```
LDA KK+1
STA NN+1
SEC
LDA KK
SBC B
STA NN
BCS FOLYT
DEC NN+1
FOLYT ...
```

NN=KK-LL (mind kétbyte-os)

```
SEC
LDA KK
SBC LL
STA NN
LDA KK+1
SBC LL+1
STA NN+1
```

NN=NN\*2

```
ASL NN
ROL NN+1
```

Megjegyzés: ASL egy bittel balra lépteti a számot, az utolsó bit helyére 0 lép be, ez kettővel való szorzást jelent. A ROL megegyezik ASL-lel, de az előző lépésben kilépett átvitelt (C jelzőből) beviszi a felső byte legalsó helyiértékére. Neggyel való szorzás a fenti két utasítás megs.i.t. Nem kettő hatványával való szorzás léptetések és hozzáadások kombinálásával lehetséges, nagyobb vagy változóban tárolt számmal való szorzásnál ciklust érdemes szervezni.

Programstruktúrák kétbyte-os számokkal:

IF NN <> 0 THEN GOTO 100

```
LDA NN
ORA NN+1
BNE CIM100
```

IF NN <> KK THEN GOTO 100

```
LDA NN
CMP KK
BNE CIM100
LDA NN+1
CMP KK+1
BNE CIM100
```

FOR NN=0 TO 4200 (4200=\$1068)

...  
NEXT

```
LDA #$69
STA VEG
LDA #$10
STA VEG+1
LDA #0
STA NN
STA NN+1
CFEJ ...
...
INC NN
BNE F1
INC NN+1
F1 LDA NN
CMP VEG
BNE CFEJ
LDA NN+1
CMP VEG+1
BNE CFEJ
```

Ugy érezzük, ennyi építőelem ismertetése már elegendő ahhoz, hogy az olvasó felismerje a bonyolultabb algoritmusok képzését ezekből. Erdemes programozni az ilyen egyszerű utasítás-sorozatok felhasználásával még akkor is, ha az ezekből összeállított program nem a leghatékonyabb, hiszen a működő programot utólag könnyebb hangolni, mint egy bonyolult, elemeiben is új megoldást levezetni.

A fentiek illusztrálására lássunk egy aránylag hosszú, de igen egyszerű programot, amely az assembly programozásban néhány gyakran előforduló építőkövet használ és egyes részfeladataival szintén sűrűn találkozunk. A program memóriadumpot ír ki a képernyőre a monitorok megszokott stílusában, egy sorba 8 byte hexadecimális és ASCII értékét is kiírva. A sor elején a sorban kinyomtatott első byte címét írja ki hexadecimálisan, amelyet egy kettőspont követ. Bár a feladat egyszerű, mégis aránylag sok, önmagában kerek részfeladatból áll, így pl.:

- hexadecimális érték beolvasása és konvertálása kétbyte-os binárisra,
- ciklus szervezése alsó határtól felső határig 8 lépéssel,
- bináris egy- és kétbyte-os érték konvertálása hexadecimálissá,
- kiírandó karakterekből a speciális karakterek kiszűrése,
- billentyűzetről olvasás,
- képernyőre írás.

Utóbbi két részfeladattal egy kicsit előreszaladunk a következő fejezet témakörébe, de megértésük most sem okoz nehézséget. Lássuk először a program teljes szövegét:

```

100      ;DUMP PROGRAM -KEZDOCIME 49170=$C012
110 CR    =$00
120 SPACE =$20
130 KERDO =$3F
140 CHRIN  =$FFCF
150 CHROUT =$FFD2
160      **$C000
170 N      =$59
180 NK     =$5B
190 MUT    .BYTE 0
200 ALAP   .BYTE 0
210 HX     .TEXT "0123456789ABCDEF"
220      ;*****
230 KEZDET JSR ERTEK    ;KEZDOCIM BEOLVASASA
240      BNE HIBA      ;HIBA VOLT
250      LDA N         ;ELSO CIM
260      STA NK        ;EZ AZ ALSO HATAR -> NK-BA
270      LDA N+1
280      STA NK+1
290      TXA           ;VEGJEL
300      CMP #CR
310      BEQ KIIR      ;HA AZ ELSO ERTEK CR-REL VEGZODO TT, NINCS MASIF
320      JSR ERTEK     ;FELSO HATAR BEOLVASASA
330      BNE HIBA      ;HIBA VOLT
340 KIIR   JSR DUMP
350 VEGE   RTS         ;VISSZA A BASIC-HEZ
360      ;*****

```



```

370 ENTER LDA #0 ;ALAP-0 JELZI, HOGY AZ ELSO SZAMJEGY KOVETKEZIK
380 STA ALAF
390 SEC JSR CHRIN ;LEOLVASSUK AZ ELEJEROL A BEVEZETO SPACE TIT
400 CMP #CR ;HA ROGTON CR JOTT, HIBA
410 BEQ SYNTAX
420 CMP #SPACE ;HA SPACE, OLVASSUK A KOVETKEZOT
430 BEQ SPC
440 BNE VIZSG ;HA NEM, EZ MAR ERTEKES JEGY
450 CIKLUS JSR CHRIN ;ERTEK KOVETKEZO KARAKTERE
460 CMP #CR ;HA CR, VEGE AZ ERTEKNEK
470 BEQ ENDE
480 CMP #SPACE ;HA SPACE, AKKOR IS
490 BEQ ENDE
500 VIZSG CMP #0 ;HA < "0" HIBAS
510 BMI SYNTAX
520 CMP #9 ;HA >="0" ES <="9" AKKOR SZAMJEGY
530 BMI SZAM
540 BEQ SZAM
550 CMP #'A ;HA < "A" AKKOR HIBA
560 BMI SYNTAX
570 CMP #'G ;HA >= "G" AKKOR HIBA
580 BPL SYNTAX
590 BETU SEC ;BETU JOTT
600 SBC #'A-10 ;KIVONJUK "A" KODJATK HELYI JUNK 10-ET
610 JMP OSSZEG ;ES HOZZAADJUK AZ EDDIGIEKET
620 SZAM SEC ;SZAM JOTT
630 SBC #0 ;KIVONJUK "0" ASCII KODJAT, IGY KONVERTALJUK
635 ;BINARISSA
640 OSSZEG LDX ALAP ;ELSO JEGY
650 BNE OCIK ;NEM, AKKOR ELTOLJUK AZ EDDIGIEKET EGY HELYI-
655 ;ERTEKKEL
660 STA N ;ELSO JEGY N-BE
670 LDA #0
680 STA N+1 ;FELSO BYTE 0
690 LDX #4 ;JELOLJUK ALAPBAN, HOGY MAR NEM ELSO JEGYROL VAN SZO
700 STX ALAP
710 JMP CIKLUS ;ES OLVAASSUK A KOVETKEZO KARAKTERT
720 OCIK ASL N ;A MAR BEOLVASOTT SZAMRESZT SZOROZZUK 16-TAL
730 ROL N+1 ;AMI EGY HEXA HELYIERTEKNEK FELEL MEG
740 DEX
750 BNE OCIK
760 OSSZF CLC ;HOZZAADJUK AZ UJ ERTEKET N-HEZ
770 ADC N
780 STA N
790 BCC 01
800 INC N+1
810 01 JMP CIKLUS ;ES OLVAASSUK A KOVETKEZO KARAKTERT
820 ENDE TAX ;VEGJEL X-BE
830 LDA #0 ;NEM VOLT HIBA
840 RTS
850 SYNTAX LDA #1 ;HIBAS VEG ERTEK SZUBRUTINBAN
860 RTS
870 ;* * * * *

```

```

880 HIBA    LDA #KERDO    ;HIBAJELZEST ADUNK KI ES VEGE A DALNAK
890        JSR CHROUT
900        LDA #CR
910        JSR CHROUT
920        JMP VEGE      ;PROGRAM VEGE
930        ;* * * * *
940 DUMP    LDA #CR      ;SOREMELES
950        JSR CHROUT
960 KOVSOR  JSR EGYSOR   ;KEZDOCIM NK, VEGCIM N
970 E1      LDA N+1      ;HA NK>N AKKOR VEGE
980        CMP NK+1
990        BMI DUVEG
1000       BNE D1        ;EGYEBKENT MEG EGY SORT KIIRUNK
1010       LDA N
1020       CMP NK
1030       BMI DUVEG
1040       BEQ DUVEG
1050 D1     CLC
1060       LDA NK        ;NK-T MEGNOVELJUK B-CAL
1070       ADC #B
1080       STA NK
1090       BCC KOVSOR
1100       INC NK+1
1110       JMP KOVSOR
1120 DUVEG  RTS         ;DUMP SZUBROUTIN VEGE
1130       ;* * * * *
1140 EGYSOR  JSR CIMKI   ;EGY DUMP SOR KIIRASA
1150       JSR HEXAKI
1160       JSR ASCIKI
1170       LDA #CR      ;VEGERE SOREMELES
1180       JSR CHROUT
1190       RTS
1200       ;* * * * *
1210 CIMKI  LDA NK+1     ;SOR KEZDOCIMENЕК KIIRASA
1220       JSR BYTEKI   ;FELSO BYTE
1230       LDA NK
1240       JSR BYTEKI   ;ALSO BYTE
1250       LDA #' '     ;KETTOSPONT
1260       JSR CHROUT
1270       LDA #SPACE   ;SZOKOZ
1280       JSR CHROUT
1290       RTS
1300       ;* * * * *
1310 HEXAKI LDY #0       ;A HEXADECEMALIS DUMP KIIRASA
1320 HCIK   LDA (NK),Y   ;KOVETKEZO BYTE
1330       JSR BYTEKI   ;HEXABAN OUTPUTRA
1340       LDA #SPACE   ;PLUSZ EGY SZOKOZ
1350       JSR CHROUT
1360       INY
1370       CPY #B
1380       BMI HCIK     ;HA Y<B KOVETKEZOT
1390       RTS
1400       ;* * * * *

```

```

1410 ABCIKI LDY #0 ; A KARAKTERES DUMP KIIRASA
1420 ACIK LDA (NK),Y ; KOVETKEZO BYTE
1430 CMP #SPACE ; HA < SPACE VAGY
1440 BMI EXTRA
1450 CMP #$7F ; >127, AKKOR AKARMI LEHET, EZERT -> EXTRA
1460 BEQ ADUT
1470 BPL EXTRA
1480 ADUT JSR CHROUT ; EGYEBKENT UGY, AHOGY VAN KIIRJUK
1490 INY
1500 CPY #8
1510 BMI ACIK ; HA Y<8 AKKOR KOVETKEZOT
1520 RTS
1530 EXTRA LDA #' ; SPECIALIS KARAKTER HELYETT "."-OT IRUNK KI
1540 JMP ADUT
1550 ; * * * * *
1560 BYTEKI PHA ; EGY BYTE-OT HEXABAN KIIR - MEGJEGYEZZUK A VEREMBEN
1570 LSR A ; ELOSZOR A MAGASABB HELYIERTÉKU HEXAJEGYET IRJUK KI
1580 LSR A ; EZERT ELOSZTJUK 16-TAL
1590 LSR A
1600 LSR A
1610 TAX
1620 LDA HX,X ; A MEGFELELO KARAKTERT
1630 JSR CHROUT ; KIIRJUK
1640 PLA ; EREDETI SZAM VEREMBOL
1650 AND #$0F ; ALSO JEGY
1660 TAX
1670 LDA HX,X ; A MEGFELELO KARAKTERT
1680 JSR CHROUT ; KIIRJUK
1690 RTS

```

Most vegyük sorra a program utasításait:

160. sor - A programot a \$C000-ra helyeztük, mivel a kazetta-puffer számára túl hosszú

230 - 250. sor - Ez a tulajdonképpeni főprogram.  
Feladatai a következők:  
Beolvas egy értéket (kezdőcím), áttölti NK-ba, majd ha nincs vége még a sornak, beolvas még egy értéket (végcím). Ezután a DUMP szubrutin segítségével kiírja a memoriadumpot.  
Az ERTEK beolvasó szubrutin A-ban nem 0 értéket ad vissza hiba esetén, ekkor a HIBA címkére kell elugrani.  
Most lássuk a szubrutinokat!

370 - 440. sor - Az ERTEK szubrutin feladata az inputban soronkövetkező hexadecimális érték beolvasása a kétbyte-os N változóba. Az első ciklus addig olvas, amíg szokóztól különböző karaktert nem talál. Így tüntetjük el a bevezető space-ket. Ha eközben egy CR (sor vége) karakter jön be, az hibának számít, SYNTAX-ra megyünk.

- 450 - 580. sor - A beolvasó ciklus első része.  
Ez a ciklus nagyon hasonlít az előzőhöz, de itt az értékes jegyek feldolgozása a feladat. A szökőz és a CR lezárja a számot, ekkor ENDE címkére ugrunk. A hexadecimális jegyként értelmezhető számjegyek (SZAM) és az A-tól F-ig terjedő betűk (BETU) esetén a megfelelő címkére megyünk, ahol a beérkezett jegyet binárisra konvertáljuk, az eddigi értékhez összegezzük, majd CIKLUS-ra ugrunk vissza. Minden egyéb karakter hibának számít.
- 590 - 630. sor - A betűk ill. a számok konvertálása a megfelelő bináris értékre.
- 640 - 810. sor - A bejött hexadecimális számjegyet összegezzük az eddig beérkezett számjegyekkel.  
A gondolatmenet a következő:  
Az utoljára érkezett hexa jegyet (egyelőre) az egyesek helyiértékén állónak tekintjük. Ha ez volt az első számjegy, egyszerűen tároljuk N-ben (660-680. sor), ha viszont már nem az első, a helyzet bonyolultabb. Ekkor az eddig beérkezett értéket 4 bináris jeggyel ( $2^4=16$ ) balra toljuk (720-750. sor), majd hozzáadjuk az új értéket (760-800. sor). Azt, hogy az első számjegyről van-e szó, ALAP-ban jelöljük. Ennek értéke 0 az első jegy feldolgozásánál, később 4-re állítjuk (690-700. sor), ami a jelzön kívül a léptetési ciklus ciklusváltozójaul is szolgál. Ezt vizsgáljuk a 640-650. sorokban. A 810. sorban visszaugrunk a CIKLUS-ra, azaz olvassuk az inputból a következő karaktert.
- 820 - 840. sor - Ez az ERTEK szubrutin logikai befejezése hibátlan esetben. A visszatérési paraméterek: beolvasott végjel X-ben, 0 A-ban.
- 850 - 860. sor - Hibás esetben viszont A=1-gyel térünk vissza.
- 880 - 920. sor - A HIBA szubrutin hívása a főprogramból ( 240. és 330. sor ) történik, ha az ERTEK szubrutin hibát jelzett. Itt egy kérdőjelet és egy kocsivissza jelet írunk ki, majd a VEGE címkére (a program logikai végére) adjuk a vezérlést.

- 940-1120. sor - Ez a szubrutin végzi a memóriadump listázását, eddig csak a paraméterek beolvasásával foglalkoztunk. A DUMP ciklusban hívja az EGYSOR szubrutint, ami egy DUMP sor kiíratását végzi az NK által mutatott címtől kezdődően. Minden sor kiírása után megvizsgáljuk, hogy NK értéke nem érte-e el N értékét, ami a DUMP végcíme. Ezután megnöveljük NK-t 8-cal és kiírjuk a következő sort az EGYSOR szubrutin segítségével és így tovább.
- 1140-1190. sor - Az EGYSOR szubrutin semmi mást nem csinál, mint sorban hívja a sor kezdőcímét kiíró, a dump hexadecimális részét kiíró és a sor ASCII részét kiíró szubrutinokat, majd egy CR-rel lezárja a sort.
- 1210-1290. sor - Ez a szubrutin az NK aktuális értékét írja ki a BYTEKI szubrutin kétszeri hívásával (felső byte, alsó byte), majd egy kettőspontot és szóközt ír ki.
- 1310-1390. sor - A HEXAKI szubrutin egy ciklusban sorra veszi a sorba tartozó 8 byte-ot és a BYTEKI szubrutinnal kiírja értéküket egy-egy szóközzel együtt.
- 1410-1540. sor - Az ASCIKI nagyon hasonlít az előzőhöz a ciklus szervezésében, hiszen itt is sorra kell venni ugyanazt a 8 byte-ot, de azokat nem a BYTEKI segítségével írjuk ki. Ezek közvetlenül outputra kerülhetnek, kivéve a speciális, pl. formátum- és színvezérlő karaktereket, amelyek összezavarnák a nyomtatási képet. Az egyszerűség kedvéért (kicsit túl egyszerűsítve a problémát) speciális karakternek tekintünk (EXTRA) minden \$20-nál kisebb és \$7F-nél nagyobb értékű karaktert. Ezek helyett pont kerül az outputra (1530-1540. sor).
- 1560-1690. sor - Végül a már többször hívott BYTEKI rutin. Bináris-hexadecimális konvertálást végez úgy, hogy először a felső hexadecimális számjegyet választja le négyszeri jobbra léptetéssel és írja ki, majd az alsó számjegyet logikai AND művelettel leválasztva konvertálja és küldi outputra. A konvertáláshoz a HX tömböt használja fel, amelyben a K-adik karakter a K értékű hexadecimális számjegy.

Az ismertetett program talán egy kicsit sok szubrutint tartalmaz, ami nem a leggyorsabban futó kódot eredményezi, viszont igen hasznos mind a program megértése, mind tesztelése és esetleges módosítása szempontjából.

### 5.3 A Commodore 64 "operációs rendszere", a KERNAL

Mint már esett szó róla, a tár \$E000-tól \$FFFF-ig terjedő területen a gép alapvető programjai, a KERNAL szubrutin-gyűjteménye található. Számunkra ez úgy jelentkezik, mint a programozáshoz segítséget adó eszköz, ezért érdemes megismerkedni a **legfontosabb szubrutinok interface**-ével és rend-tésével. A felhasználó által hívható valamennyi szubrutin interface-e megtalálható a függelékben.

CHROUT - egy byte-ot ír ki az éppen aktiv output file-ra. Ha nem volt még ilyen megnyitva (CHKOUT-tal), az output a képernyőre megy.

Belépési cím: \$FFD2

Paraméterek : A - kinyomtatandó karakter ASCII kódja

Példa : LDA #\$0D

JSR CHROUT

egy kocsivissza-jelet ír ki

CHRIN - egy byte-ot olvas az éppen aktiv input file-ról. Ha nem volt még ilyen megnyitva (CHKIN-nel), a billentyűzetről olvas. Utóbbi esetben a képernyőszerkesztőt aktivizálja, amely villogtatja az aktuális pozícióban a cursort, beolvas egy logikai sort a BASIC-sorpufferbe (\$200), majd mindenegybes CHRIN hívásra egy karaktert ad át, egészen addig, amíg a puffer ki nem ürül, ekkor az egész előlről kezdődik.

Belépési cím: \$FFCF

Paraméterek : A - a beolvasott karakter ASCII kódja

Példa : JSR CHRIN

STA BYTE

CHKIN - megnyit olvasásra egy file-t. A file-sorszámot az X regiszterben várja.

Belépési cím: \$FFC6

Paraméterek : X - file-sorszám

Példa : LDX #5

JSR CHKIN

előkészíti olvasásra a korábban megnyitott 5-ös file-t

CHKOUT - megnyit írásra egy file-t. A file-sorszámot az X regiszterben várja.

Belépési cím: \$FFC9

Paraméterek : X - file-sorszám

Példa : LDX #5

JSR CHKOUT

CLRCHN - lezárja a nyitott input és/vagy output file-okat, de a file-ok nyitva maradnak.  
Belépési cím: \$FFCC  
Paraméterek : -

A fenti szubrutinok használatával igen egyszerűen valósítható meg az assembly programok input/output tevékenysége. A file-ok megnyitását és lezárását BASIC-ből vagy más KERNAL-rutinokkal végezhetjük.

A gép ROM-rutinjai úgy lettek tervezve, hogy a felhasználó könnyen beavatkozhatson működésükbe. Így pl. a felhasználó által is hívható KERNAL-rutinok többsége a memória RAM részében elhelyezett vektorokon keresztül léptethető be. Ezen vektorok átírásával a KERNAL-rutin meghívásakor a vezérlést saját programunkhoz terelhetjük és ott tetszés szerint ténykedhetünk. Igaz ez a BASIC-interpreter néhány fontosabb szubrutinjára is. Az átírható KERNAL-vektorok:

|             |   |        |
|-------------|---|--------|
| \$31A-\$31B | - | OPEN   |
| \$31C-\$31D | - | CLOSE  |
| \$31E-\$31F | - | CHKIN  |
| \$320-\$321 | - | CHKOUT |
| \$322-\$323 | - | CLRCHN |
| \$324-\$325 | - | CHRIN  |
| \$326-\$327 | - | CHROUT |
| \$328-\$329 | - | STOP   |
| \$32A-\$32B | - | GETIN  |
| \$32C-\$32D | - | CLALL  |

Lássunk egy példát a KERNAL átírására! Tegyük fel, hogy egy speciális printerünk van, amely bizonyos kódokat másként értelmez, mint a Commodore 64 belső kódrendszere, ami egyébként csak nagyjából felel meg az ASCII-nak. Így lehet ez például egy olyan nyomtató esetében, amely ékezetes betűket tud nyomtatni, de ezek a betűk nem azon karakterek kódjainak felelnek meg, amelyek lenyomásával mi rögzíteni kívánjuk őket, pl. a magyar írógépszabvány billentyűi szerint. A képernyőkódok megfelelő összerendelését a karakterképekkel könnyű megoldani, de azután még el kell végezni az átkódolást minden egyes karakter esetén, amelyet a nyomtatóra küldünk. Az alábbi kis szubrutin első belépési pontja a CHROUT-vektort írja át saját rutinunkra, amely az átkódolást végzi:

```
10 CHROUV = $326
20 CHROUT = $F1CA
30 EGYSEG=$9A
40 *=$CE00
50 BEALL LDA #SAJAT<
60 STA CHROUV
```



```

70      LDA #SAJAT>
80      STA CHROUV+1
90      RTS
100     SAJAT  PHA          ; OUTPUT BYTE STACK-BE
110      LDA EGYSEG
120      CMP #4          ; PRINTER?
130      PLA ATKOD
140      TXA            ; EREDETI BYTE
150      JMP CHROUT      ; OUTPUTRA
160     ATKOD  LDA TABLA,X ; MEGFELELO KOD
170      JMP CHROUT      ; OUTPUTRA
180     TABLA  .BYTE 0,1,2,3,4,5,6,7,8,9,$A,$B,$C,$D,$E,$F
...

```

Még egy megjegyzés a fenti programhoz: A \$9A cím az aktuális output eszköz egységszámát tartalmazza.

## 5.4 BASIC és assembly békés egymás mellett élése

A Commodore 64 gépre készülő assembly programok többsége a BASIC-interpreterrel és a BASIC-programokkal igen szoros kapcsolatot tart. Leggyakrabban olyan BASIC-kiegészítéseket programozunk assembly nyelven, amelyek a BASIC programból nehezen vagy nem elég hatékonyan megoldható feladatokat teljesítenek.

A BASIC-kiegészítések programozásához természetesen alaposan ismerni kell a ROM-ban lévő interpreter működését, memória-használatát, belepési pontjait, a BASIC program felépítését, stb. Mindezen ismereteket itt nem lenne helyenvaló és nem is szükséges felsorolni, ezek nagyrészt a Commodore BASIC-kel foglalkozó irodalom tartalmazza, ld. pl. (1), (3).

Ebben a fejezetben csak azokat az ismereteket foglaljuk össze, amelyek speciálisan az assembly nyelvű kiegészítéseket érintik és más munkákban nem, vagy nem ilyen szempontból vannak leírva. **Feltételezzük egy sor olyan ismeret meglétét, amely inkább a BASIC programozáshoz tartozik egészen a BASIC tárfelosztás és interpreter-kód mélységéig.** Ha az olvasó nem rendelkezik ezen ismeretekkel, vagy lapozza át ezt a fejezetet, vagy legyen szert ezekre az ismeretekre pl. a fent megjelölt munkákból, és csak azután tanulmányozza ezt a fejezetet.

Az egész változók (jele %) egyedi változóként 7, tömbelemként 2 byte-ot foglalnak a memóriában, de az első esetben a 7-ből 3 byte kihasználatlan. Két byte-on a változó neve és típusa, két byte-on maga az értéke van elhelyezve nem cím formában, hanem **felső byte - alsó byte sorrendben.**

Az egyedi változó tehát:

- 0. byte - név első karaktere + 128
- 1. byte - név második karaktere vagy 0+128
- 2. byte - magasabb helyiértékű (F) byte
- 3. byte - alacsonyabb helyiértékű (A) byte
- 4-6. byte - közömbös (0)

Az egész tömb szerkezete:

- 0. byte - név első karaktere + 128
- 1. byte - név második karaktere vagy 0+128
- 2-3. byte - következő tömb kezdőcíme, a tömb kezdetéhez relatív mutató!
- 4. byte - dimenziók száma (n)
- 5-től 2\*n+4-ig - az egyes dimenziók felső határai két-byte-os egész formában (azaz felső - alsó byte sorrendben).
- elemszám\*2 byte - a tömb értékei, mint az egyszerű változónál (felső-alsó byte) sorrendben.

A lebegőpontos numerikus változók számára a BASIC 2+5=7 byte-ot tart fenn, a tömbelemek számára 5-öt. A FAC - lebegőpontos akkumulátor - tárgyalásánál megtudtuk, hogy a szám ábrázolásához 5 byte-ra, valamint az előjel tárolására egy külön byte-ra van szüksége az interpreternek. A változók számára viszont elég 5 byte. **Hogy is van ez?**

A lebegőpontos(FAC formátumu) ábrázolásban először is a számot bináriszá kell alakítani, majd úgy beírni a mantissza tárolására szolgáló 1-4. byte-okba, hogy a szám első 1-ese a legelső pozícióra kerüljön. Pl. a 12 binárisan:

1100

Igy kerül a FAC-ba:

|                  | 0.        | 1.        | 2. | 3. | 4. |
|------------------|-----------|-----------|----|----|----|
| decimálisan :    | 132       | 102       | 0  | 0  | 0  |
| hexadecimálisan: | 84        | C0        | 0  | 0  | 0  |
| binárisan :      | 1000 0100 | 1100 0000 | 0  | 0  | 0  |

A 0. byte-ot, az exponenst úgy állítjuk elő, hogy az első értékes jegy előjeles távolságát a bináris ponttól (esetünkben 4) hozzáadjuk 128-hoz. A szám előjele megegyezik a FAC 5. byte-jának előjelével. A nulla kódja megegyezés szerint csupa nulla byte. A memóriaformátum már egy kicsit különbözik a FAC formátumtól, mivel annál nem tartunk fenn egy külön byte-ot az előjelnek. A fenti algoritmusból következően a FAC formájú szám 1. byte-jának 0. bitje mindig 1, kivéve, ha értéke nulla. Így ez a bit nem hordoz információt, nyugodtan helyettesíthetjük ezt a bitet a szám előjelbitjével (1, ha negatív, 0, ha pozitív). A memóriában tárolt lebegőpontos érték így különbözik egy kicsit a FAC-ban számolásra kész értéktől, töltéskor ill. tároláskor a konverzióról az interpreternek gondoskodni kell. A változók és tömbök szerkezete megegyezik az egész változók és tömbökével, csak az érték hossza mindenütt 5 byte, valamint a név változatlanul kerül tárolásra, ezzel jelezve a típust.

A karakteres változók már nem tartalmazzák a változó értéket, mivel a stringeket a BASIC interpreter egy külön összefüggő területen helyezi el a változó hosszúságú string-kezeléshez szükséges szemetgyújtás megkönnyítésére. A karakteres változó ill. tömbelem értékes része 3 byte hosszú, amelyből a 0. byte a karakterek száma, az 1-2. byte-ok pedig a string első karakterének az abszolút címe a memóriában.

A fenti számábrázolási formátumok egymásba konvertálása, a kifejezések kiértékelése, a változók és tömbelemek megkeresése, új változók deklarálása, létrehozása bonyolult prog-

ramozási feladatot jelentene, de szerencsére nem kell megcsinálni, mivel ezt megtették már az interpreter írásakor. Miért is ne használnánk ezeket a szubrutinokat assembly programunkból? Az alábbiakban az interpreter olyan belépési pontjait – és azok paraméterezését – ismertetjük, amelyek talán leghasznosabbak a BASIC-kiegészítések írásakor:

- FRMEVL - A soronkövetkező BASIC kifejezés kiértékelése és az eredmény elhelyezése FAC-ban. Ha a kifejezés karakteres volt, FAC 3-4. byte-ja (\$64-\$65) a string 3 byte-os leírójára mutat.  
Belépési pont: \$AD9E
  
- INTFAC - Az A (alsó byte) és az Y (felső byte) regiszterekben lévő kétbyte-os előjeles egész szám konvertálása lebegőpontos formára és elhelyezése FAC-ban.  
Belépési pont: \$B391
  
- YTOFAC - Az Y regiszter tartalmának konvertálása lebegőpontos formára és elhelyezése FAC-ban.  
Belépési pont: \$B3A2
  
- MEMFAC - Az A (alsó byte) és Y (felső byte) egy memóriacímét tartalmaz. A címen lévő 5 byte-os lebegőpontos szám átvitele FAC-ba (formátumváltással).  
Belépési pont: \$BBA2
  
- STRFAC - String konvertálása lebegőpontos formára és az érték elhelyezése FAC-ban. A string hosszát az A regiszter, kezdőcímet \$22-\$23 tartalmazza.  
Belépési pont: \$B7B5
  
- MEMFC2 - Ugyanaz, mint MEMFAC, de az eredmény a második lebegőpontos akkumulátorba, FAC2-be kerül (\$69-\$6E). A \$6F címen lévő byte 0. bitje 1 lesz, ha FAC és FAC2 előjele különböző. Ez a byte az ún. előjel-összehasonlítási jelző (SCF - sign comparasion flag).  
Belépési pont: \$BABC
  
- FACINT - FAC tartalmát kétbyte-os előjeles egészzé konvertálja \$64-\$65-be teszi. FAC-ot elrontja. Az egész szám felső byte - alsó byte formájú.  
Belépési pont: \$B1AA
  
- FACADR - Ugyanaz, mint FACINT, de az eredmény kétbyte-os előjel nélküli egész (0-65535) cím formátumban (alsó byte - felső byte) a \$14-\$15 címre kerül. FAC-ot elrontja.  
Belépési pont: \$B7F7

- FACMEM - MEMFAC fordítottja. A memóriaterület címét X (also byte) és Y (felső byte) regiszterekben várja el a rutin.  
Belépési pont: \$BBD7
- FACSTR - A FAC tartalma karaktersorozattá konvertálva a \$100 címre kerül. A stringet egy 0 byte zárja.  
Belépési pont: \$BDDD
- FC2FAC - FAC2 másolása FAC-ba.  
Belépési pont: \$BBFC
- FACFC2 - FAC másolása FAC2-be.  
Belépési pont: \$BCOF
- FPLUS - FAC=FAC+FAC2  
A rutin hívása előtt az SCF-et megfelelően állítani kell (LDA \$66 / EOR \$6E / STA \$6F) és az A regiszterbe a FAC exponenset kell tölteni (LDA \$61).  
Belépési pont: \$BB6A
- FMINUS - FAC=FAC-FAC2  
Belépési pont: \$B853
- FMULT - FAC=FAC\*FAC2  
Az előkészítés ugyanabból áll, mint FPLUS-nál.  
Belépési pont: \$BA30
- FDIV - FAC=FAC2/FAC  
Előkészítés, mint FPLUS-nál.  
Belépési pont: \$BB12
- FPower - FAC=FAC2^FAC  
Előkészítés, mint FPLUS-nál.  
Belépési pont: \$BF7B
- ADRESS - A BASIC szövegben következő változó címének elökeresése (lehet tömbelem is). A változó címet (ahol a neve megtalálható) a \$5F-\$60-ba írja. Tömbelem megadása esetén ez a tömb nevére (kezdetére) és \$59-\$5A a tömb első értékére mutat. Az A (alsó byte) és az Y (felső byte) regiszterek magára az értékre (string esetén a hárombyte-os leíróra) mutatnak. Ugyanezt a címet tartalmazza \$47-\$48 is.  
\$0D egyenlő \$FF-fel, ha a változó karakteres típusu, egyébként 0 (string flag). \$0E egyenlő \$80-nal, ha a változó egész típusu, egyébként 0 (integer flag).  
Belépési pont: \$B08B

LINEIN - Egy logikai sor beolvasása a képernyőszerkesztő segítségével a BASIC input pufferbe (\$200). A sor maximális hossza 80 byte, a karaktersorozatot bináris egy 0 byte zárja le. Belépési pont: \$A560

LINOUT - Egy karaktersorozat kiírása a képernyőre az aktuális pozíciótól kezdve. Az A (alsó byte) és Y (felső byte) regiszterekben várja a szubrutin a karaktersorozat kezdőcímét. Belépési pont: \$AB1E

Lássunk egy példát, amelyik a fenti szubrutinok közül talán a legbonyolultabbat, az ADRESS-t használja. A program feladata a más BASIC-interpreterek által elfogadott ERASE utasítás megvalósítása.

```

10 ; ERASE SZUBRUTIN TOMB TORLESEHEZ
20 ; HIVASA SYS 49152,A$(0,0...), AHOL AZ INDEXEK
30 ; ERTEKE TETSZOLEGES. A TOMBNEK MAR DEKLARALTNAK
40 ; KELL LENNI. A SZUBRUTIN LEFUTASA UTAN A TOMB
50 ; UJRA DIMENZIONALHATO
60 ADRESS=$B08B
70 TVEGE = $31
80 CHRGET=$73
90 TCIM = $5F
100 KOV = $47
110 * = $C000
120 ERASE JSR CHRGET ; TERMINATOR
130 JSR ADRESS ; TOMB CIME TCIMBE
140 LDY #2 ; KOVETKEZO TOMB CIME
150 LDA (TCIM),Y
155 ADC TCIM
160 STA KOV ; KOV-BE
170 INY
180 LDA (TCIM),Y
185 ADC TCIM+1
190 STA KOV+1
200 LDY #0
210 HASONL LDA KOV ; KOV=TVEGE?
220 CMP TVEGE
230 BNE CIKLUS
240 LDA KOV+1
250 CMP TVEGE+1
260 BEQ CIKVEG ; IGEN --> VEGERE
270 CIKLUS LDA (KOV),Y ; EGY BYTE MASOLASA
280 STA (TCIM),Y
290 INC KOV ; MUTATOK INKREMENTALASA
300 BNE F1
310 INC KOV+1
320 F1 INC TCIM
330 BNE HASONL
340 INC TCIM+1
350 JMP HASONL

```

```

360 CIKVEG   LDA   TCIM           ; TOMBVEG-POINTER CSOKKENTESE
370          STA   TVEGE
380          LDA   TCIM+1
390          STA   TVEGE+1
400          RTS                ; VISSZA A BASIC-HEZ

```

Nézzuk lépésről-lépésre!

- 120. sor - Az elválasztó vessző átlépése.
- 130. sor - A tömböt megkeresi a BASIC-területen és kezdőcímét \$5F-be (TCIM) teszi.
- 140-190. sor - A tömb leírásából kiemeljük a következő tömb címét (2.-3. byte) és KDV-be rakjuk. A feladat tulajdonképpen az ettől a címtől a BASIC tömbök végéig (TVEGE=\$31-\$32) terjedő terület visszacsúsztatása TCIM-re. Ezzel eltüntetjük a paraméterként kapott tömböt.
- 200. sor - Y regiszter nullázása indexeléshez.
- 210-260. sor - Megvizsgáljuk, hogy KDV értéke elérte-e már TVEGE értéket. Ha igen, vege a másolásnak.
- 270-280. sor - Egy byte másolása.
- 290-350. sor - KDV és TCIM növelése és visszatérés a ciklus elejére (HSONL).
- 360-400. sor - TCIM most oda mutat - pontosabban az utolsó utáni byte-ra - ahova a tömbök hátralévő részét átmásoltuk. Ez a cím lesz a TVEGE - mutató új értéke. Ezután visszatérünk a BASIC-hez.

A BASIC-kel való kapcsolattartás lényeges része a BASIC program és az assembly szubrutin vezérlésátadásának és paraméterezésének kérdése.

Már megismerkedtünk ezen módszerek egy részével, de azért foglaljuk a legfontosabbakat össze:

Assembly rutin hívása BASIC-ből:

- SYS BASIC utasítással,
- USR függvényként,
- beszurásként.

Az utóbbiról még nem volt szó, ez tulajdonképpen a BASIC-interpreter módosítása, kiegészítése. Mint a KERNAL-rutinok némelyike, a BASIC-interpreter néhány rutinja is úgy lett tervezve, hogy **beléptetésük egy RAM-ban tárolt vektoron át történik**, amelynek átírásával a ROM-ban lévő szubrutin helyett saját programunkhoz kerül a vezérlés. Ezután természetesen visszatérhetünk az interpreter bármely pontjára, akár a kiváltott BASIC-interpreter rutinra is. A RAM-ban tárolt vektorok és a hozzájuk tartozó szubrutinok:

|             |   |                                                                                                                            |
|-------------|---|----------------------------------------------------------------------------------------------------------------------------|
| \$300-\$301 | - | Hibauzenet kiírása                                                                                                         |
| \$302-\$303 | - | BASIC melegindítás                                                                                                         |
| \$304-\$305 | - | Egy sor átváltása interpreter kódra (tokenizálás)                                                                          |
| \$306-\$307 | - | Interpreter kód átváltása LIST típusu sorrá                                                                                |
| \$308-\$309 | - | A következő BASIC utasítás végrehajtása                                                                                    |
| \$30A-\$30B | - | Numerikus term (konstans vagy változó) bevitele FAC-ba. FRMEVL hívja a kifejezés minden tagjánál (nemigen érdemes átírni). |

A beszurások onnan kapták nevüket, hogy a BASIC-interpreter kódvégrehajtási folyamatába szurjuk be azokat és segítségükkel lehetőségünk nyílik annak eldöntésére, hogy nekünk, vagy az interpreternek szóló parancsrol van-e szó. Leggyakoribb a \$308-\$309 vektor átírása. Ha ezt saját szubrutinunk belépési pontjára állítjuk át, minden BASIC utasítás előtt nekünk adódik a vezérlés. Ekkor \$7A-\$7B a soronkövetkező karakterre mutat a BASIC program szövegében. Ha az utasítás vizsgálata azt az eredményt adja, hogy nekünk szól, hajtjuk végre és gondoskodjunk arról, hogy \$7A-\$7B-t a következő utasítás elejére állítsuk. Ellenkezo esetben egyszeruen ugorjunk JMP-pal a \$A7E4 címre, ami a \$308-\$309 alapértéke. Ha az interpreter-kódra váltó rutin elé is beszurást teszünk (a \$304-\$305 vektor átírásával), lehetőségünk nyílik arra, hogy saját parancsainkat is felismerjük és egybyte-os kódra - tokenre cseréljük. Ekkor az utasítás végrehajtásnál lényegesen felgyorsul, egyetlen byte ellenőrzésére csökken a beszurás feladata. Ilyenkor persze nemcsak az interpreter-kódra váltó, hanem az arról olvasható szövegre konvertáló (vektor: \$306-\$307) rutin elé is beszurást illik tenni, hogy a LIST olvasható programot írjon ki.

#### Paraméterátadás BASIC-ból assembly-be:

- **SYS utasítással hívott szubrutinnak** a már ismert módon, a FRMEVL interpreter rutin segítségével adhatunk át tetszőes szerinti számú és típusu paramétert, kifejezéseket is. Másik megoldás a regiszterek használata. Ha BASIC programból SYS-szel hívunk egy gépikódu címet, az interpreter a rutin meghívása előtt az A,X,Y,P regisztereket rendre a \$30C,\$30D,\$30E,\$30F byte-ok tartalmával tölti fel, majd a rutin visszatérése után a regisztereket ide menti el. Ezeket a memóriacímeket a BASIC program PEEK-kel és POKE-kal elérheti, így ezeken keresztül egyszerű paraméterátadás valósítható meg.



- **USR-rel** hívott rutin egy numerikus értéket kap a FAC-ban és ugyanott egy numerikus értéket ad is vissza. Ezenkívül csak memórián keresztül adható át paraméter.
- **BASIC-beszúrás** esetén rajtunk áll, hogy értelmezzük a nekünk szóló utasítást és az interpreter mely részeit használjuk fel az utasítás végrehajtása során.

#### Visszatérés a BASIC-interpreterbe:

mindhárom típusu assembly-kiegészítés **RTS-sel** kell, hogy végetérjen, hibás esetektől eltekintve, amikor a megfelelő hibaágra kell ugranunk.

## 5.5 Gyorsabban, messzebbre, magasabbra...

A sportból kölcsönzött fejezetcím jól jelképezi azt, hogy elsősorban milyen típusu feladatok megoldására érdemes használni az assembly programozás lehetőségeit. Igen fontos, szinte elkerülhetetlen alkalmazása azokban az esetekben, amikor a magas szintu nyelven – általában BASIC-ben – irt megoldáshoz kepest a végrehajtási idő jelentős felgyorsítása a cél, vagy a probléma természetéből adódóan, vagy egyszerű ergonómiai szempontok alapján. Az elsőre példa szinte **valamennyi komolyabb játékprogram**, ahol a grafika animációja, a hangeffektusok kezelése, a felhasználói beavatkozásokra történő reagálás olyan végrehajtási sebességet követel meg, amely BASIC-ből, de jószérivel a többi magas szintu nyelvből sem biztosítható. Az utóbbira inkább az adatfeldolgozás köréből hozhatunk fel példákat. Magasszintu nyelven megírva bizonyos elemi adatkezelési tevékenységek (**rendezés, változtatás, keresés, összefuttatás**) annyira lassúak, hogy a felhasználótól nem követelhető meg, hogy kivárja azokat. Különösen igaz ez az **adattábaziskezelésre**, magasszintu file-kezelésre, ill. ezek visszakereső és file-módosító funkcióira.

Mivel az első típusu feladatok (játék-software) inkább egyedi, speciális problémákat vetnek fel, inkább az utóbbi területről választunk szemléltető példát. Emellett szól az is, hogy míg egy játékprogram valamely érdekesebb részének elemzése nem nyújtana kis változtatással felhasználható építőköveket olvasóink további programozási munkájához, addig az itt következő programok részleteinek vagy egészenek alkalmazása véleményünk szerint hasznos lehet.

Az egyik legidőigényesebb adatfeldolgozási művelet a **rendezés**. Anélkül, hogy a rendezési algoritmusok elméleti kérdéseibe túlságosan belemennénk, bevezetőben elégedhetetlen néhány szót szólni a rendezés módszereiről.

Erre a feladatra igen sok algoritmus ismeretes. Itt természetesen csak **a legegyszerűbb algoritmusok** bemutatásáról lehet szó, de ezek a mikrogépes feladatok legfeljebb néhány ezer rekord rendezési igénynek tökéletesen megfelelnek, amellett gyorsan és egyszerűen programozhatók. A legegyszerűbb még használható algoritmus **a buborék-módszer**. Ennek során minden egyes cikluslépésben kikeressük a még rendezetlen lista legnagyobb elemét s egyszersmind a legutolsó pozícióba juttatjuk.

Ezt úgy végezzük, hogy az egymás után következő elemeket páronként összehasonlítjuk egymással. Ha a második elem kisebb az előtteállónál, felcseréljük őket. Végig folytatva a legnagyobb elem az utolsó helyre vándorol, tehát elegendő a következő ciklust az öt megelőző elemig folytatni, (az már eggyel rövidebb lesz), amíg csak el nem érünk az első, most már biztosan legkisebb elemhez. A módszer onnan kapta a nevét, hogy a kisebb elemek a rendezés folyamán úgy szállnak

feljebb és feljebb, mint a légbuborékok a folyadékban. A hatékonyságot egy egyszerű trükkel fokozhatjuk, ha egy jelzőt használunk arra, hogy egy ciklusban történt-e elemcsere, vagy sem. Ha nem történt, nyugodtan befejezettnek nyilváníthatjuk a rendezést, hiszen ez azt jelenti, hogy minden elem kisebb-egyenlő az utánakövetkezőnél, tehát a lista rendezett. Erre a javított algoritmusra közlünk egy programot, amely egy egész típusú BASIC-tömb rendezését végzi, természetesen helyben:

```

10 ; BUBOREK MODSZER EGESZ TOMB RENDEZESERE
20 ; AZ ELEMeket ELOJEL NELKULI EGESZ SZAMOKNAK TEKINTJUK
30 ;
40 ; HIVASA : SYS49152,R%(0) , AHOL R% A RENDEZNI KIVANT TOI
50 ;
55 CHRGET=$73
60 MERET = $FB
70 K = $FD
80 AKT = $5A
90 KOV = $5C
100 JELZO = $FF
110 TCIM = $5F
120 ADRESS= $B08B
125 * = $C000
127 JSR CHRGET ; VESSZO
130 JSR ADRESS ; TOMB KEZDOCIM TCIM-BE
140 LDY #5 ; TOMB MERETE
150 LDA (TCIM),Y
160 STA MERET+1 ; MERET-BE
170 INY
180 LDA (TCIM),Y
190 STA MERET
200 KULSO LDA TCIM ; KOV=TCIM+7 --> 0. ELEM CIME
210 CLC
220 ADC #7
230 STA KOV
240 LDA TCIM+1
250 ADC #0
260 STA KOV+1
270 LDA #0
280 STA K ; CIKLUSVALTOZO NULLAZASA
290 STA K+1
300 STA JELZO ; CSERE-JELZO LEGYEN HAMIS
310 BELSO LDA KOV ; AKT=KOV : KOV=KOV+2
320 STA AKT
330 CLC
340 ADC #2
350 STA KOV
360 LDA KOV+1
370 STA AKT+1
380 BCC U2
390 INC KOV+1
400 U2 LDY #1 ; OSSZEHASONLITAS
410 SEC
420 LDA (KOV),Y

```

```

430          SBC (AKT),Y
440          DEY
450          LDA (KOV),Y
460          SBC (AKT),Y
470          BCS NEMCSE          ; NAGYOBB
480          LDX #1              ; VOLT CSERE
490          STX JELZO
500 CSERE   LDA (AKT),Y
510          TAX
520          LDA (KOV),Y
530          STA (AKT),Y
540          TXA
550          STA (KOV),Y
560          INY
570          CPY #2
590          BNE CSERE
600 NEMCSE  INC K                ; K=K+1
610          BNE U3
620          INC K+1
630 U3      LDA K                ; HA K=MERET, VEGE
640          CMP MERET           ; A BELSO CIKLUSNAK
650          BNE BELSO
660          LDA K+1
670          CMP MERET+1
680          BNE BELSO
690          LDA JELZO          ; VOLT CSERE?
700          BEQ VEGE           ; NEM
710          LDA MERET          ; MERET=MERET-1
720          SEC
730          SBC #1
740          STA MERET
750          BCS U4
760          DEC MERET+1
770 U4      LDA MERET            ; HA MERET=0, VEGE
780          BNE KULSO
790          LDA MERET+1
800          BNE KULSO
810 VEGE    RTS                  ; VISSZA A BASIC-HEZ

```

Nézzük a programot sorról-sorra!

130. sor - A tömbnév paraméter átvétele BASIC-ból és a tömbcím kiszámítása az ADRESS interpreter-rutin segítségével.

140-190. sor - A tömb deklarált méretének kiemelese a tömb leírójából és a MERET változóba írása.

200-260. sor - A külső ciklus feje. KOV mutatót a tömb 0. elemére állítjuk.

- 270-300. sor - Nullázzuk a csere-jelzöt és a ciklusváltó-  
zót, amely a belső ciklus hosszáig nő majd.
- 310-390. sor - A belső ciklus feje. Az AKT mutatót a KOV-  
vel tesszük egyenlővé, a KOV az AKT utáni  
elemre állítódik.
- 400-470. sor - Az egymást követő két szám kivonásával el-  
döntjük, melyik nagyobb. Ha a második,  
nincs csere.
- 480-490. sor - Ha az első, a jelzőben jelöljük a cserét,  
majd
- 500-590. sor - megcseréljük a két elemet.
- 600-620. sor - Eggyel növeljük a ciklusváltozó értékét.
- 630-680. sor - Ha kisebb, mint az aktuális ciklusméret,  
folytatjuk a vizsgálatot,
- 690-700. sor - ha elértük a ciklus végét, megvizsgáljuk,  
volt-e csere. Ha nem, kész a rendezés.
- 710-800. sor - Eggyel csökkentjük a MERET-et, s ha nullá-  
ra ért, véget vetünk a rendezésnek, egyéb-  
ként folytatjuk a külső ciklust.
810. sor - Vissza a BASIC-hez, a tömb rendezett.

A fenti módszer nagyon jól használható **nem túl nagy méretű** tömbök rendezéséhez. Ha azonban a méret növekedni kezd, és különösen ha nem egész számok, hanem karaktersorozatok rendezését kell végezni vagy beletörődünk abba, hogy néhány száz elem rendezése már percekig igényel, vagy új módszer után nézünk.

Ha nem akarjuk teljesen eldobni a buborék-rendezés programját és a vele szerzett tapasztalatokat, érdemes egy olyan eljárást keresni, amelyik annak továbbfejlesztése. A buborék-rendezésről már tudjuk, hogy akkor hatékony, ha aránylag kevés elemet kell rendezni és komolyan felgyorsítható, ha sejtjük, hogy a lista közel rendezett és alkalmazzuk a csere jelzöt. Ezekből az információkból már kinálkozik a **keverő-rendezés (merge sort)** módszere, ami abból áll, hogy az elemeket sok kis halmazra osztjuk, ezeket buborék-módszerrel rendezzük, a kis halmazokat nagyobb, részben rendezett listákká futtatjuk össze, ezeket buborék-rendezzük s.i.t., míg el nem érjük a teljes tömb méretét.

Hogy a példaprogram ne hasonlítson olyan nagyon az előzőre, ezt a módszert egy **karakters BASIC tömb helybenrendezésén** mutatjuk be:

```

100 ; MERGE S/ASS
110 ; HIVASA BASIC-BOL: SYS 50688,A$(X),N AHOL A$(X) A
115 ; KEZDOELEM, N AZ ELEMENK SZAMA
120 MERET = $FB
130 K = $FD
140 AKT = $57
150 KOV = $59
160 JELZO = $FF
170 STR1 = $5B
180 STR2 = $5D
190 HOSSZ1 = $5F
200 HOSSZ2 = $60
210 KITEVO = $4E
220 NAGYSZ = $4F
230 L = $51
240 TAROL = $26
250 CHRGET = $73
260 FRMEVL = $AD9E
270 ADRESS = $B08B
280 GETADR = $B7F7
290 TCIM = $47
300 Z1 = $14
310 *=5Q688
320 JSR CHRGET ; TERMINATOR BE
330 JSR ADRESS ; AZ ELZO RENDEZENDO TOMBELEM CIME A KCIMBE
332 LDA TCIM ; TCIM ELMENKES KCIM-BE
334 STA KCIM
336 LDA TCIM+1
338 STA KCIM+1
340 JSR CHRGET ; MASODIK VESSZO
350 JSR FRMEVL ; MERET FAC-BA
360 JSR GETADR ; CIM FORMABAN Z1-BE
370 LDA Z1 ; MERET-BE
380 STA MERET
390 LDA Z1+1
400 STA MERET+1
410 LDA #1 ; NAGYSZ=1
420 STA NAGYSZ
430 LDA #0 ; KITEVO=0
440 STA KITEVO
450 STA NAGYSZ+1
460 HATVC INC KITEVO ; KITEVO=KITEVO+1
470 ASL NAGYSZ ; NAGYSZ=2*NAGYSZ
480 ROL NAGYSZ+1
490 SEC
500 LDA NAGYSZ ; IF NAGYSZ<MERET THEN HATVC
510 SBC MERET
520 LDA NAGYSZ+1
530 SBC MERET+1
540 BCC HATVC
550 KULSO LSR NAGYSZ+1 ; NAGYSZ=NAGYSZ/2
560 ROR NAGYSZ
570 SEC
580 LDA MERET ; K=MERET-NAGYSZ
590 SBC NAGYSZ
600 STA K

```

```

610      LDA MERET+1
620      SBC NAGYSB+1
630      STA K+1
640 KOZEPS LDA #0.          ; JELZO=0
650      STA JELZO
660      STA L              : L=0
670      STA L+1
680      LDA KCIM           ; AKT=KCIM
690      STA AKT
700      LDA KCIM+1
710      STA AKT+1
720      LDA NAGYSG+1      ; TAROL=NAGYSAG
730      STA TAROL
740      LDA NAGYSG        ; KOV=3*NAGYSG+AKT
750      ASL A
760      ROL TAROL
770      CLC
780      ADC NAGYSG
790      ADC AKT
800      STA KOV
810      LDA TAROL
820      ADC NAGYSG+1
830      ADC AKT+1
840      STA KOV+1
850 BELSO  LDY #0          ; HOSSZ1 ES HOSSZ2
860      LDA (AKT),Y
870      STA HOSSZ1
880      LDA (KOV),Y
890      STA HOSSZ2
900      INY
910      LDA (AKT),Y      ; STR1 ES STR2
920      STA STR1
930      LDA (KOV),Y
940      STA STR2
950      INY
960      LDA (AKT),Y
970      STA STR1+1
980      LDA (KOV),Y
990      STA STR2+1
1000     LDY #0
1010 HASON  LDA (STR2),Y   ; IF (STR2)<(STR1) THEN CSERE
1015     ; ELSE NEMCSE
1020     CMP (STR1),Y
1030     BCC CSERE
1040     BNE NEMCSE
1050     INY
1060     CPY HOSSZ1
1070     BEQ NEMCSE
1080     CPY HOSSZ2
1090     BEQ CSERE
1100     BNE HASON
1110 HOPP1  BNE KOZEPS    ; VISSZAUGRASOK
1120 HOPP2  BNE KULSO
1130 CSERE  LDY #2        ; VOLT CSERE
1140      STY JELZO
1150 CSEREC LDA (AKT),Y   ; CSERECIKLUS

```

```

1160      TAX
1170      LDA (KOV),Y
1180      STA (AKT),Y
1190      TXA
1200      STA (KOV),Y
1210      DEY
1220      BPL CSEREC
1230 NEMCSE INC L           ; L=L+1
1240      BNE U1
1250      INC L+1
1260 U1    LDA AKT           ; AKT=AKT+3 -> KOVETKEZO ELEM
1270      CLC
1280      ADC #3
1290      STA AKT
1300      BCC U2
1310      INC AKT+1
1320 U2    LDA KOV           ; KOV=KOV+3 -> KOVETKEZO ELEM
1330      CLC
1340      ADC #3
1350      STA KOV
1360      BCC U3
1370      INC KOV+1
1380 U3    LDA K             ; IF L<K THEN GOTO BELSO
1390      CMP L
1400      BNE BELSO
1410      LDA K+1
1420      CMP L+1
1430      BNE BELSO
1440      LDA JELZO           ; IF NOT JELZO THEN GOTO JELZOH
1450      BEQ JELZOH
1460      SEC
1470      LDA K               ; K=K-1
1480      SBC #1
1490      STA K
1500      BCS U4
1510      DEC K+1
1520      LDA K
1530 U4    BNE HOPP1          ; IF K<>0 THEN KOZEPS
1540      LDA K+1
1550      BNE HOPP1
1560 JELZOH DEC KITEVO        ; KITEVO=KITEVO-1
1570      BNE HOPP2          ; IF KITEVO<>0 THEN KULSO
1580      RTS                ; VEGE, VISSZA A HIVOHOZ
1590 KDIM  .WORD 0

```

A fenti program algoritmusá egy kicsit összetettebb az előzőnél. Megpróbálunk kulcsot adni a megértéséhez, de igazán csak úgy könnyű megérteni, ha egy rövid (8-10 elemű) lista rendezési példájára az olvasó maga követi papíron vagy akár számítógépen, lépésről-lépésre végrehajtva a programot. Ezt a könyv példáit tartalmazó lemezen lévő MERGEBAS-TEST program igen nagymértékben elősegíti. Az alábbiakban utasításról -utasításra értelmezzük a programot:



- 320-330. sor - A már ismert módon átvesszük a hívó SYS utasítás első paraméterét, a kezdőelemet ill. annak címét a KCIM-ben. Megjegyzendő, hogy ez a szubrutin nem az egész tömböt rendezi, hanem csak a megadott elemmel kezdődő, megadott számú elemet. A szubrutin nem végez méretellenőrzést, amiből igen komoly programhibák származhatnak. Szép feladat a 400. sor után ilyen ellenőrzés beiktatása, amely BAD SUBSCRIPT ERROR hibát ad a tömb méretének túllépése esetén.
- 340-400. sor - A második paraméter, a rendezendő elemek számának átvétele és elhelyezése kétbyte-os egészként a MERET nevű változóban.
- 410-450. sor - A NAGYSG-ot 1-re, a kitevőt 0-ra állítjuk.
- 460-540. sor - Megkeressük azt a 2-es hatványt, amelynek értéke nagyobb vagy egyenlő a mérettel. Ezt úgy végezzük, hogy ciklusban addig szorozzuk a NAGYSG-ot 2-vel, amíg kisebb méretnél. A ciklusok számát KITEVO-ben számoljuk.
- 550-560. sor - Ez a legkülső ciklus feje. Ebben a ciklusban (amely KITEVO-szor fut le) egy rendezési menet zajlik le. A részlistákat teljesen lerendezzük buborék-eljárással. Az egyes menetekben annyi részre osztjuk a teljes halmazt, amennyi NAGYSG értéke és ezeket a részeket egymástól teljesen függetlenül rendezzük, bár a program szervezésében ez összefonódik. Mivel NAGYSG értékét a ciklus elején rögtön felére csökkentjük, az első menetben a listák száma kettőnek az a hatványa, ami meg éppen kisebb MERET-nél. Ezután minden legkülső cikluslépésben felére csökken - éppen ezen két utasítás végrehajtásával - NAGYSG értéke, egészen addig, míg az utolsó menetben egy lesz, azaz a teljes tömböt egészben kell lerendezni.
- 580-630. sor - A K ciklusváltozó a középső ciklusban, értéke MERET-NAGYSG, lényegében az utolsó részlista első eleme és a tömb vége közötti távolságot adja meg.
- 640-650. sor - A középső ciklus kezdete. A ciklus egy-egy lépése a buborék-rende-zés külső cikluslépésinek felel meg. Egy-egy cikluslépésben - a belső ciklus szerint - valamennyi listát rendezi párhuzamosan. Első ténykedése a csere-jelző 0-ra állítása - szerepe ugyanaz, mint a buborék-eljárásnál.

- 660-670. sor - A belső ciklusváltozó 0-zása.
- 680-710. sor - Az összehasonlítási ciklus első eleme a teljes lista első eleme lesz, ennek címét KCIM-ből vesszük.
- 720-840. sor - Itt kezdődik a legbelső ciklus, ami annyi lépésből áll, amennyi K értéke, ezzel valamennyi listában megteesszük a soronkövetkező összehasonlítást. A ciklusban az első teendő a két összehasonlítandó bejegyzés hosszának kiemelése a leírókból és a HOSSZ1 és HOSSZ2 munkaváltozókba tárolása.
- 910-990. sor - Ugyanezt tesszük a leíró második részével, a karakterláncok kezdőcímével is. Ezeket STR1 és STR2, a nullás lapon elhelyezkedő pointerekbe írjuk.
1000. sor - Az Y regisztert nullázzuk az indirekt indexelt címzéshez. Az Y regiszter így tehát a stringek első (0.) byte-jára mutat.
- 1010-1100. sor - Ebben a ciklusban történik a karaktersorozatok összehasonlítása. Ha ennek n.-edik lépésében
- a második string-beli karakter kisebb (a tőle balra lévők idáig egyenlők voltak!), a második karaktersorozat kisebb, csere szükséges;
  - az első karaktersorozat aktuális karaktere a kisebb, az első string kisebb, nem cserélünk;
  - az első karakterlánc véget ért, definíció szerint kisebb, nem kell cserélni;
  - a második karakterlánc ért véget, ő a kisebb, cserélni kell;
  - a karakterek egyenlők, mindkét lánc még tart, folytatjuk az összehasonlítást.
- 1110-1120. sor - Ez a két utasítás a program végéről továbbítja az ugrásokat, mivel a cél már távolabb van az ugró utasítástól 128 byte-nál.
- 1130-1140. sor - Ide kerül a vezérlés csere esetén. Először a csere-jelzőt állítjuk 0-tól különböző értékre.
- 1150-1220. sor - A karaktersorozatok cseréje helyett mindössze a hárombyte-os leírókat cseréljük ki. Ezt ebben a ciklusban tesszük, egy menetben egy byte-ot. Átmeneti tárolásra az X regisztert használjuk.

- 1230-1250. sor - Ha nem cserélünk, rogtón, ha igen a csere után lépünk ide. Megnöveljük eggyel a ciklusváltozó értékét.
- 1260-1370. sor - Mindkét elemre mutató pointer értékét 3 byte-tal, egy elemmel előre állítjuk, vagyis áttérünk a következő részlista következő elemére.
- 1380-1430. sor - Megvizsgáljuk, hogy elértük-e a belső ciklus végét, ha nem, folytatjuk BELSO-nél.
- 1440-1450. sor - Ha nem volt csere, a külső ciklus végellenőrzéséhez ugrunk, mivel ez azt jelenti, hogy a részlisták már rendezettek.
- 1460-1520. sor - Csökkentjük a középső ciklus változóját is.
- 1530-1550. sor - Ha nem csökkent 0-ra, az egyes részrendezések következő lépésére térünk át, azaz a középső ciklus folytatódik.
1560. sor - Csökkentjük a kitevő értékét.
1570. sor - Ha nem 0, visszatérünk a külső ciklus elejére és most feleannyi részlistát fogunk rendezni. Ha 0, már az egész lista rendezett, tehát a  $2^0=1$  részlistát is lerendeztük.
1580. sor - Visszatérés a BASIC programba.

Ez az eljárás lényegesen jobb eredményt ad, mint az előző, még nagyobb elemszámok esetén is. Tájékoztatásul: 1000 db, átlagosan 8 hosszúságú véletlenszerűen generált string rendezése kb. 21 másodpercet igényelt, ami nem rossz eredmény.

A programot házi feladatként érdemes továbbfejleszteni. Például a teljes stringek helyett csak adott pozíciótól kezdődő adott számú, vagy előre definiált terminátorig terjedő karaktereket tekinteni rendezési kulcsnak és eszerint teljes rekordokat rendezni. Ennél sokkal egyszerűbb feladat lehet az itt adott növekvő rendezés helyett paraméterezhetően csökkenő vagy növekvő rendezést használni. Ugyancsak érdekes feladatot jelenthet az azonos kulcsok közül az első kivételével mind elhagyása és a keletkezett tömb tömörítése, valamint a megmaradt elemek számának kimenő paraméterként való visszaadása, stb. Az ilyen feladatok megoldását, persze teszteléssel együtt, amolyan kis házi vizsgaként is lehet értékelni. Aki pl. a legutolsó átalakítást meg tudja oldani, nyugodt lehet, elsajátította a könyvben tárgyalt ismereteket.

## 5.6 Nagy feladatok megközelítése

Míg az eddigiekben jobbára csak rövid, kiegészítőként alkalmazott programcskákról, szubrutinokról volt szó, könyvünk utolsó fejezetében szeretnénk néhány jótanácsot adni **utavalóul** azoknak, akik ennél nagyobb fába vágják fejszájüket és önálló, nagyterjedelmű software-terméket kívánnak létrehozni.

A nagyobb programozási feladatok tulajdonképpen egy hosszabb rendszerfejlesztési tevékenységnek csak kis részét képezik és ennek megfelelően kell kezelni őket. A rendszerfejlesztési tevékenységnek – anélkül, hogy most szervezési fejtegetésekbe bocsátkoznánk – egy lehetséges felosztás szerint a következő fő részei vannak:

1. Helyzetfelmérés, a rendszerrel szemben támasztott input és output igények, valamint hatékonysági követelmények megismerése.
2. Külső specifikáció, vagy nagyvonalú rendszerterv készítése, amely dönt a megoldásban felhasználható hardware és software elemekről, pontosan leírja a kész rendszerrel szemben támasztott követelményeket, a termék elkészülését bizonyító átvételi teszt mélységéig.
3. Részletes rendszerterv kidolgozása, amely mező mélységig specifikálja az adatállományokat – adatfeldolgozás esetén – és algoritmus – blokkdiagram vagy egyéb kodifikált algoritmus-leírás – mélységig a programot. Ezáltal programozásra alkalmas szervezési dokumentációként kezelhető.
4. Programozási tevékenységek, amelyek programtervezésből, tesztelési terv elkészítéséből és magából a kódolásból állanak.
5. Programtesztelési munkálatok, amelynek során először modulonként, majd a kész modulokat programmá integrálva programonként, végül a teljes rendszerre teljesítjük a tesztelési tervben foglaltakat.
6. A rendszeren lefuttatjuk az átvételi tesztet, hatékonysági vizsgálatot végzünk, javaslatot teszünk kódoptimalizálásra és visszatérünk a 4. ponthoz, vagy befejezettenek nyilvánítjuk a terméket.
7. A szervíztevékenység folyamatos ellátása.

Ezen lépések bármelyikében olyan döntésekre kényszerülhetünk, hogy valamelyik korábbi tevékenységi pontra visszatérve az azóta elkészült dokumentumok és termékek átdol-

gozásával ill. újra elkészítésével következetesen folytassuk a munkamenetet. Az olyan - **elegendően nagyméretű - programok, amelyek nem a fenti vagy ahhoz hasonló szervezett módon készülnek, megbizhatatlan kódot eredményeznek**, amelynek utólagos szerviz-ráfordításai messze meghaladják azt a költséget, amelyet a helyesen végzett fejlesztő tevékenység igényelt volna.

A teljes rendszerfejlesztési vertikum veghezvitele a programozási szaktudáson kívül egyéb képzettséget, elsősorban **szervezői ismereteket** is igényel. Mivel a mikroszámítógépek területén igen gyakori az egyszemélyes, vagy igen kis csoport által végzett munka, különösen fontos, hogy az ilyen körülmények között dolgozó szakemberek bizonyos fokig univerzálisak legyenek. Gyakran lebecsülik a kisgépes fejlesztői tevékenységet, s míg a nagygépeknel szinte mindig betartják a fenti, vagy ahhoz hasonló munkaszervezés lépcsőfokait és az egyes lépésekben megfelelő képzettségű specialistákat alkalmaznak, addig a kisgépes rendszerek készítését kis munkának tekintik. Ez igen nagy és igen veszélyes tévedés, mivel a mikrogepek általában korlátozott software-ellátottsága és néha nagyságrendekkel kisebb hatékonysági mutatói mellett igen magas felhasználói igényeket megkívánó rendszereket kell megvalósítani. Valószínűleg ennek a lebecsülésnek a következménye az is, hogy feltűnően sok rossz minőségű programterméket kínálnak szerte a világban személyi számítógépekre.

Ha a nagyobb rendszerek fejlesztésében szükséges programozási feladatokra koncentrálunk, igen fontos már **a tervezési szakaszban**

- a lehető legkisebb részekre - akár assembly-makró szintig - dekomponálni a feladatot és definiálni az egyes elemek pontos interface-et. Általában ezen tevékenység közben kiderülnek a negyvonalu terv súlyos hibái és kialakul, mintegy elveszti szabadságfokát a modul algoritmus.
- Meg kell tervezni a tesztelés pontos tervét modulonként, mivel nagyobb mennyiségű teszteletlen kód átlátása megoldhatatlan neheztségeket okozhat. Nem szabad sajnálni a fáradságot a modulok számára megfelelő teszt-ágy készítésére, amely hűen szimulálja a többi modul interface-szerinti viselkedését. A tesztelési tervnek pontosan tartalmaznia kell a tesztek várható eredményeit is és ennek ellenőrzését sem árt teszt-ágyra bízni.

#### **A kódolási szakaszban**

- nem szabad eltérni a programtervtől, teljesíthetetlenség esetén vissza kell menni a tervezési szakaszhoz.

## A tesztelés során

- pontosan ragaszkodni kell a megtervezett tesztelési folyamathoz,
- csak tökéletesen letesztelt modulokat integráljunk - azokból sem túl sokat - nagyobb egységekké. Soha ne spóroljuk meg a teszt-ágy készítését magunk vagy mások által leteszteltnek hitt modulok alkalmazásával!

A teljes munka alatt pontosan **dokumentáljuk** tevékenységünket, bátran használjuk a számítógépet ennek támogatásában! Minél több szervezési, tervezési, programozási, tesztelési dokumentációt őrzünk meg, annál könnyebb lesz a hibák felderítése. Ne sajnáljuk a dokumentálásra fordított időt akkor sem, ha mi magunk vagyunk programunk megrendelője, szervezője, tervezője, kódolója, tesztelője és végfelhasználója egy személyben! **Semmi sem tud olyan ismeretlen lenni, mint a fél éve általunk írt program!**

Végül, ha jóminőségű terméket akarunk létrehozni, vegyük figyelembe a személyi számítógépekre írt programok néhány speciális **külső minőségtényezőjét:**

- A képernyő legyen jól szervezett, áttekinthető, nem túl zsufolt. Kerüljük a buta, BASIC programokra jellemző rollozást.
- A képernyőn mindig történjen valami. Bármilyen bonyolult feladatot is old meg programunk, a felhasználó egy perc után feltétlenül - néha már hamarabb - gyanakodni kezd.
- Segítsük a gyakorlatlan felhasználót jól áttekinthető, érthető pontokból álló menüvel, könnyen kiváltható help-funkciókkal.
- Ne gátoljuk a gyakorlott felhasználót mindezen - hosszantartó - elemekkel, adjunk lehetőséget számára rövid, sokatmondó részekből álló összetett parancsok kiadására.
- Bátran - de mértéktelenül - használjuk ki a gép speciális lehetőségeit, grafikát, hangot, joystick-et, fényceruzát, stb. Ugyanakkor adjuk meg a lehetőségét annak is, hogy az ilyen speciális eszközökkel nem rendelkező felhasználók is használhassák a programot.

A fenti felhasználói kapcsolat-elemek példaszerű megvalósításával találkozhat az olvasó a **SUPERBASE 64** programban (más kérdés, hogy adatbáziskezelésre mennyire alkalmas).

Végül a mikroszámítógépes rendszerfejlesztési tevékenység talán legfontosabb pontjára szeretnénk felhívni az olvasó figyelmét. Ne próbáljunk minden felmerülő feladatot feltétlenül megoldani Commodore 64 segítségével. Azokról az igényekről, amelyek, úgy tűnik, éppenhogy megoldhatók a géppel, menetközben kiderül, hogy sokkal ágasbogasabbak és ilyenkor már nehéz abbahagyni a munkát. Így csak lábujjhegyen álló, könnyen összeomló rendszerek készíthetők, amelyek akkor mondják fel a szolgálatot, amikor a legkellemetlenebb. **Használjuk a Commodore 64-et arra, amire való** és sok örömünk lesz mind a programok fejlesztésében, mind alkalmazásában.

A FUGGELEK  
COMMODORE 64 KODTABLA

| HELY | HELY | HELY    | HELY | HELY   | HELY   | HELY | HELY |
|------|------|---------|------|--------|--------|------|------|
| 0    | 00   |         | @    | SORVEG | BRK    | 0    | 00   |
| 1    | 01   |         | A    |        | ORA/IX | 1    | 01   |
| 2    | 02   |         | B    |        |        | 2    | 02   |
| 3    | 03   |         | C    |        |        | 3    | 03   |
| 4    | 04   |         | D    |        |        | 4    | 04   |
| 5    | 05   |         | E    |        | ORA/N  | 5    | 05   |
| 6    | 06   |         | F    |        | ASL/N  | 6    | 06   |
| 7    | 07   |         | G    |        |        | 7    | 07   |
| 8    | 08   |         | H    |        | PHP    | 8    | 08   |
| 9    | 09   |         | I    |        | ORA/KV | 9    | 09   |
| 10   | 0A   |         | J    |        | ASL/KV | 10   | 0A   |
| 11   | 0B   |         | K    |        |        | 11   | 0B   |
| 12   | 0C   |         | L    |        |        | 12   | 0C   |
| 13   | 0D   | RETURN  | M    |        | ORA/A  | 13   | 0D   |
| 14   | 0E   |         | N    |        | ASL/BE | 14   | 0E   |
| 15   | 0F   |         | O    |        |        | 15   | 0F   |
| 16   | 10   |         | P    |        | BPL/A  | 16   | 10   |
| 17   | 11   | CRSR LE | Q    |        | ORA/IY | 17   | 11   |
| 18   | 12   | RVS ON  | R    |        |        | 18   | 12   |
| 19   | 13   | CRSHOME | S    |        |        | 19   | 13   |
| 20   | 14   | DELETE  | T    |        |        | 20   | 14   |
| 21   | 15   |         | U    |        | ORA/NX | 21   | 15   |
| 22   | 16   |         | V    |        | ASL/NX | 22   | 16   |
| 23   | 17   |         | W    |        |        | 23   | 17   |
| 24   | 18   |         | X    |        | CLC    | 24   | 18   |
| 25   | 19   |         | Y    |        | ORA/HY | 25   | 19   |
| 26   | 1A   |         | Z    |        |        | 26   | 1A   |
| 27   | 1B   |         | [    |        |        | 27   | 1B   |
| 28   | 1C   |         | £    |        |        | 28   | 1C   |
| 29   | 1D   | CRSJOB  | ]    |        | ORA/AX | 29   | 1D   |
| 30   | 1E   |         | ↑    |        | ASL/AX | 30   | 1E   |
| 31   | 1F   |         | ←    |        |        | 31   | 1F   |
| 32   | 20   |         |      |        | JSR    | 32   | 20   |
| 33   | 21   | !       | !    | !      | AND/IX | 33   | 21   |
| 34   | 22   | "       | "    | "      |        | 34   | 22   |
| 35   | 23   | #       | #    | #      |        | 35   | 23   |
| 36   | 24   | \$      | \$   | \$     | BIT/N  | 36   | 24   |
| 37   | 25   | %       | %    | %      | AND/Z  | 37   | 25   |
| 38   | 26   | &       | &    | &      | ROL/Z  | 38   | 26   |
| 39   | 27   | /       | /    | /      |        | 39   | 27   |
| 40   | 28   | (       | (    | (      | PLP    | 40   | 28   |
| 41   | 29   | )       | )    | )      | AND/KV | 41   | 29   |
| 42   | 2A   | *       | *    | *      | ROL/BE | 42   | 2A   |
| 43   | 2B   | +       | +    | +      | ROL/Z  | 43   | 2B   |
| 44   | 2C   | ,       | ,    | ,      | BIT/A  | 44   | 2C   |
| 45   | 2D   | -       | -    | -      | AND    | 45   | 2D   |
| 46   | 2E   | .       | .    | .      | ROL    | 46   | 2E   |
| 47   | 2F   | /       | /    | /      |        | 47   | 2F   |
| 48   | 30   | 0       | 0    | 0      | BMI    | 48   | 30   |
| 49   | 31   | 1       | 1    | 1      | AND/IY | 49   | 31   |



| 11:24 | 11:25 | 11:26 | 11:27 | 11:28 | 11:29  | 11:30 | 11:31 |
|-------|-------|-------|-------|-------|--------|-------|-------|
| 50    | 32    | 2     | 2     | 2     |        | 50    | 32    |
| 51    | 33    | 3     | 3     | 3     |        | 51    | 33    |
| 52    | 34    | 4     | 4     | 4     |        | 52    | 34    |
| 53    | 35    | 5     | 5     | 5     | AND/NX | 53    | 35    |
| 54    | 36    | 6     | 6     | 6     | ROL/NX | 54    | 36    |
| 55    | 37    | 7     | 7     | 7     |        | 55    | 37    |
| 56    | 38    | 8     | 8     | 8     | SEC    | 56    | 38    |
| 57    | 39    | 9     | 9     | 9     | AND/AY | 57    | 39    |
| 58    | 3A    | :     | :     | :     |        | 58    | 3A    |
| 59    | 3B    | :     | :     | :     |        | 59    | 3B    |
| 60    | 3C    | <     | <     | <     |        | 60    | 3C    |
| 61    | 3D    | =     | =     | =     | AND/AX | 61    | 3D    |
| 62    | 3E    | >     | >     | >     | ROL/AX | 62    | 3E    |
| 63    | 3F    | ?     | ?     | ?     |        | 63    | 3F    |
| 64    | 40    | @     | @     | @     | RTI    | 64    | 40    |
| 65    | 41    | A     | A     | A     | EOR/IX | 65    | 41    |
| 66    | 42    | B     | B     | B     |        | 66    | 42    |
| 67    | 43    | C     | C     | C     |        | 67    | 43    |
| 68    | 44    | D     | D     | D     |        | 68    | 44    |
| 69    | 45    | E     | E     | E     | EOR/N  | 69    | 45    |
| 70    | 46    | F     | F     | F     | LSR/N  | 70    | 46    |
| 71    | 47    | G     | G     | G     |        | 71    | 47    |
| 72    | 48    | H     | H     | H     | PHA    | 72    | 48    |
| 73    | 49    | I     | I     | I     | EOR/KV | 73    | 49    |
| 74    | 4A    | J     | J     | J     | LSR/BE | 74    | 4A    |
| 75    | 4B    | K     | K     | K     |        | 75    | 4B    |
| 76    | 4C    | L     | L     | L     | JMP/A  | 76    | 4C    |
| 77    | 4D    | M     | M     | M     | EOR/A  | 77    | 4D    |
| 78    | 4E    | N     | N     | N     | LSR/A  | 78    | 4E    |
| 79    | 4F    | O     | O     | O     |        | 79    | 4F    |
| 80    | 50    | P     | P     | P     | BVC    | 80    | 50    |
| 81    | 51    | Q     | Q     | Q     | EOR/IV | 81    | 51    |
| 82    | 52    | R     | R     | R     |        | 82    | 52    |
| 83    | 53    | S     | S     | S     |        | 83    | 53    |
| 84    | 54    | T     | T     | T     |        | 84    | 54    |
| 85    | 55    | U     | U     | U     | EOR/NX | 85    | 55    |
| 86    | 56    | V     | V     | V     | LSR/NX | 86    | 56    |
| 87    | 57    | W     | W     | W     |        | 87    | 57    |
| 88    | 58    | X     | X     | X     | CLI    | 88    | 58    |
| 89    | 59    | Y     | Y     | Y     | EOR/AY | 89    | 59    |
| 90    | 5A    | Z     | Z     | Z     |        | 90    | 5A    |
| 91    | 5B    |       |       |       |        | 91    | 5B    |
| 92    | 5C    |       |       |       |        | 92    | 5C    |
| 93    | 5D    |       |       |       | EOR/AX | 93    | 5D    |
| 94    | 5E    |       |       |       | LSR/AX | 94    | 5E    |
| 95    | 5F    |       |       |       |        | 95    | 5F    |
| 96    | 60    |       |       |       | RTS    | 96    | 60    |
| 97    | 61    |       |       |       | ADC/IX | 97    | 61    |
| 98    | 62    |       |       |       |        | 98    | 62    |
| 99    | 63    |       |       |       |        | 99    | 63    |

| 0000 | 0000 | 00000000 | 00000000 | 00000000 | 00000000 | 0000 | 0000 |
|------|------|----------|----------|----------|----------|------|------|
| 100  | 64   |          |          |          |          | 100  | 64   |
| 101  | 65   |          |          |          | ADC/N    | 101  | 65   |
| 102  | 66   |          |          |          | ROR/N    | 102  | 66   |
| 103  | 67   |          |          |          |          | 103  | 67   |
| 104  | 68   |          |          |          | PLA      | 104  | 68   |
| 105  | 69   |          |          |          | ADC/KV   | 105  | 69   |
| 106  | 6A   |          |          |          | ROR/BE   | 106  | 6A   |
| 107  | 6B   |          |          |          |          | 107  | 6B   |
| 108  | 6C   |          |          |          | JMP/I    | 108  | 6C   |
| 109  | 6D   |          |          |          | ADC/A    | 109  | 6D   |
| 110  | 6E   |          |          |          | ROR/A    | 110  | 6E   |
| 111  | 6F   |          |          |          |          | 111  | 6F   |
| 112  | 70   |          |          |          | BVS      | 112  | 70   |
| 113  | 71   |          |          |          | ADC/IY   | 113  | 71   |
| 114  | 72   |          |          |          |          | 114  | 72   |
| 115  | 73   |          |          |          |          | 115  | 73   |
| 116  | 74   |          |          |          |          | 116  | 74   |
| 117  | 75   |          |          |          | ADC/NX   | 117  | 75   |
| 118  | 76   |          |          |          | ROR/NX   | 118  | 76   |
| 119  | 77   |          |          |          |          | 119  | 77   |
| 120  | 78   |          |          |          | SEI      | 120  | 78   |
| 121  | 79   |          |          |          | ADC/AY   | 121  | 79   |
| 122  | 7A   |          |          |          |          | 122  | 7A   |
| 123  | 7B   |          |          |          |          | 123  | 7B   |
| 124  | 7C   |          |          |          |          | 124  | 7C   |
| 125  | 7D   |          |          |          | ADC/AX   | 125  | 7D   |
| 126  | 7E   |          |          |          | ROR/AX   | 126  | 7E   |
| 127  | 7F   |          |          |          |          | 127  | 7F   |
| 128  | 80   |          |          | END      |          | 128  | 80   |
| 129  | 81   |          |          | FOR      | STA/IX   | 129  | 81   |
| 130  | 82   |          |          | NEXT     |          | 130  | 82   |
| 131  | 83   |          |          | DATA     |          | 131  | 83   |
| 132  | 84   |          |          | INPUT#   | STY/N    | 132  | 84   |
| 133  | 85   |          |          | INPUT    | STA/N    | 133  | 85   |
| 134  | 86   |          |          | DIM      | STX/N    | 134  | 86   |
| 135  | 87   |          |          | READ     |          | 135  | 87   |
| 136  | 88   |          |          | LET      | DEY      | 136  | 88   |
| 137  | 89   |          |          | GOTO     |          | 137  | 89   |
| 138  | 8A   |          |          | RUN      | TXA      | 138  | 8A   |
| 139  | 8B   |          |          | IF       |          | 139  | 8B   |
| 140  | 8C   |          |          | RESTORE  | STY/A    | 140  | 8C   |
| 141  | 8D   |          |          | GOSUB    | STA/A    | 141  | 8D   |
| 142  | 8E   |          |          | RETURN   | STX/A    | 142  | 8E   |
| 143  | 8F   |          |          | REM      |          | 143  | 8F   |
| 144  | 90   |          |          | STOP     | BCC      | 144  | 90   |
| 145  | 91   | CRSRFEL  |          | ON       | STA/IY   | 145  | 91   |
| 146  | 92   | RVS OFF  |          | WAIT     |          | 146  | 92   |
| 147  | 93   | CLR      |          | LOAD     |          | 147  | 93   |
| 148  | 94   | INST     |          | SAVE     | STY/NX   | 148  | 94   |
| 149  | 95   |          |          | VERIFY   | STA/NX   | 149  | 95   |

| 1000 | 1000 | 1000    | 1000 | 1000   | 1000   | 1000 | 1000 |
|------|------|---------|------|--------|--------|------|------|
| 150  | 96   |         |      | DEF    | STX/NY | 150  | 96   |
| 151  | 97   |         |      | POKE   |        | 151  | 97   |
| 152  | 98   |         |      | PRINT# | TYA    | 152  | 98   |
| 153  | 99   |         |      | PRINT  | STA/AY | 153  | 99   |
| 154  | 9A   |         |      | CONT   | TXS    | 154  | 9A   |
| 155  | 9B   |         |      | LIST   |        | 155  | 9B   |
| 156  | 9C   |         |      | CLR    |        | 156  | 9C   |
| 157  | 9D   | CRSRBAL |      | CMD    | STA/AX | 157  | 9D   |
| 158  | 9E   |         |      | SYS    |        | 158  | 9E   |
| 159  | 9F   |         |      | OPEN   |        | 159  | 9F   |
| 160  | A0   |         |      | CLOSE  | LDY/KV | 160  | A0   |
| 161  | A1   |         |      | GET    | LDA/IX | 161  | A1   |
| 162  | A2   |         |      | NEW    | LDX/KV | 162  | A2   |
| 163  | A3   |         |      | TAB    |        | 163  | A3   |
| 164  | A4   |         |      | TO     | LDY/N  | 164  | A4   |
| 165  | A5   |         |      | FN     | LDA/N  | 165  | A5   |
| 166  | A6   |         |      | SPOK   | LDX/N  | 166  | A6   |
| 167  | A7   |         |      | THEN   |        | 167  | A7   |
| 168  | A8   |         |      | NOT    | TRV    | 168  | A8   |
| 169  | A9   |         |      | STEP   | LDA/KV | 169  | A9   |
| 170  | AA   |         |      | +      | TAX    | 170  | AA   |
| 171  | AB   |         |      | -      |        | 171  | AB   |
| 172  | AC   |         |      | *      | LDY/A  | 172  | AC   |
| 173  | AD   |         |      | /      | LDA/A  | 173  | AD   |
| 174  | AE   |         |      |        | LDX/A  | 174  | AE   |
| 175  | AF   |         |      | AND    |        | 175  | AF   |
| 176  | B0   |         |      | OR     | BCS    | 176  | B0   |
| 177  | B1   |         |      |        | LDA/IY | 177  | B1   |
| 178  | B2   |         |      | =      |        | 178  | B2   |
| 179  | B3   |         |      |        |        | 179  | B3   |
| 180  | B4   |         |      | SGN    | LDY/NX | 180  | B4   |
| 181  | B5   |         |      | INT    | LDA/NX | 181  | B5   |
| 182  | B6   |         |      | ABS    | LDX/NX | 182  | B6   |
| 183  | B7   |         |      | USR    |        | 183  | B7   |
| 184  | B8   |         |      | FRE    | CLV    | 184  | B8   |
| 185  | B9   |         |      | POS    | LDA/AY | 185  | B9   |
| 186  | BA   |         |      | SQR    | TSX    | 186  | BA   |
| 187  | BB   |         |      | RND    |        | 187  | BB   |
| 188  | BC   |         |      | LOG    | LDY/AX | 188  | BC   |
| 189  | BD   |         |      | EXP    | LDA/AX | 189  | BD   |
| 190  | BE   |         |      | COS    | LDX/AY | 190  | BE   |
| 191  | BF   |         |      | SIN    |        | 191  | BF   |
| 192  | C0   |         |      | TAN    | CPY/KV | 192  | C0   |
| 193  | C1   |         |      | ATN    | CMP/IX | 193  | C1   |
| 194  | C2   |         |      | PEEK   |        | 194  | C2   |
| 195  | C3   |         |      | LEN    |        | 195  | C3   |
| 196  | C4   |         |      | STR\$  | CPY/N  | 196  | C4   |
| 197  | C5   |         |      | VAL    | CMP/N  | 197  | C5   |
| 198  | C6   |         |      | ASC    | DEC/N  | 198  | C6   |
| 199  | C7   |         |      | CHR\$  |        | 199  | C7   |

| 0000 | 0000 | 0000 | 0000 | 0000    | 0000   | 0000 | 0000 |
|------|------|------|------|---------|--------|------|------|
| 200  | C8   |      |      | LEFT\$  | INY    | 200  | C8   |
| 201  | C9   |      |      | RIGHT\$ | CMP/KV | 201  | C9   |
| 202  | CA   |      |      | MID\$   | DEX    | 202  | CA   |
| 203  | CB   |      |      |         |        | 203  | CB   |
| 204  | CC   |      |      |         | CPY/A  | 204  | CC   |
| 205  | CD   |      |      |         | CMP/A  | 205  | CD   |
| 206  | CE   |      |      |         | DEC/A  | 206  | CE   |
| 207  | CF   |      |      |         |        | 207  | CF   |
| 208  | D0   |      |      |         | BNE    | 208  | D0   |
| 209  | D1   |      |      |         | CMP/IY | 209  | D1   |
| 210  | D2   |      |      |         |        | 210  | D2   |
| 211  | D3   |      |      |         |        | 211  | D3   |
| 212  | D4   |      |      |         |        | 212  | D4   |
| 213  | D5   |      |      |         | CMP/NX | 213  | D5   |
| 214  | D6   |      |      |         | DEC/NX | 214  | D6   |
| 215  | D7   |      |      |         |        | 215  | D7   |
| 216  | D8   |      |      |         | CLD    | 216  | D8   |
| 217  | D9   |      |      |         | CMP/AY | 217  | D9   |
| 218  | DA   |      |      |         |        | 218  | DA   |
| 219  | DB   |      |      |         |        | 219  | DB   |
| 220  | DC   |      |      |         |        | 220  | DC   |
| 221  | DD   |      |      |         | CMP/AX | 221  | DD   |
| 222  | DE   |      |      |         | DEC/AX | 222  | DE   |
| 223  | DF   |      |      |         |        | 223  | DF   |
| 224  | E0   |      |      |         | CPX/KV | 224  | E0   |
| 225  | E1   |      |      |         | SBC/IX | 225  | E1   |
| 226  | E2   |      |      |         |        | 226  | E2   |
| 227  | E3   |      |      |         |        | 227  | E3   |
| 228  | E4   |      |      |         | CPX/N  | 228  | E4   |
| 229  | E5   |      |      |         | SBC/N  | 229  | E5   |
| 230  | E6   |      |      |         | INC/N  | 230  | E6   |
| 231  | E7   |      |      |         |        | 231  | E7   |
| 232  | E8   |      |      |         | INX    | 232  | E8   |
| 233  | E9   |      |      |         | SBC/KV | 233  | E9   |
| 234  | EA   |      |      |         | NOP    | 234  | EA   |
| 235  | EB   |      |      |         |        | 235  | EB   |
| 236  | EC   |      |      |         | CPX/A  | 236  | EC   |
| 237  | ED   |      |      |         | SBC/A  | 237  | ED   |
| 238  | EE   |      |      |         | INC/A  | 238  | EE   |
| 239  | EF   |      |      |         |        | 239  | EF   |
| 240  | F0   |      |      |         | BEQ    | 240  | F0   |
| 241  | F1   |      |      |         | SBC/IY | 241  | F1   |
| 242  | F2   |      |      |         |        | 242  | F2   |
| 243  | F3   |      |      |         |        | 243  | F3   |
| 244  | F4   |      |      |         |        | 244  | F4   |
| 245  | F5   |      |      |         | SBC/NX | 245  | F5   |
| 246  | F6   |      |      |         | INC/NX | 246  | F6   |
| 247  | F7   |      |      |         |        | 247  | F7   |
| 248  | F8   |      |      |         | SED    | 248  | F8   |
| 249  | F9   |      |      |         | SBC/AY | 249  | F9   |

| Address | Hex | ASCII | Comment | Address | Hex |
|---------|-----|-------|---------|---------|-----|
| 250     | FA  |       |         | 250     | FA  |
| 251     | FB  |       |         | 251     | FB  |
| 252     | FC  |       |         | 252     | FC  |
| 253     | FD  |       | SBC, AX | 253     | FD  |
| 254     | FE  |       | INC, AX | 254     | FE  |
| 255     | FF  |       |         | 255     | FF  |

## B. függelék

### A MOS 6510 utasításkészlete

#### 1. Címzési módok szerint rendezve:

Abszolút:  $nn$

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEC/EOR/INC/JMP/JSR/LDA/  
LDX/LDY/LSR/ORA/ROL/ROR/SBC/STA/STX/STY

Közvetlen:  $\#n$

ADC/AND/CMP/CPX/CPY/EOR/LDA/LDX/LDY/ORA/SBC

Nullás lapu:  $\phi n$

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEC/EOR/INC/LDA/LDX/LDY/  
LSR/ORA/ROL/ROR/SBC/STA/STX/STY

Beleértétt:  $R$

ASL/BRK/CLC/CLD/CLI/CLV/DEX/DEY/INX/INY/LSR/NOP/PHA/  
PHP/PLA/PLP/ROL/ROR/RTI/RTS/SEC/SED/SEI/TAX/TAY/TSX/  
TXA/TXS/TYA

Relatív:  $\pm n$

BCC/BCS/BEQ/BMI/BNE/BPL/BVC/BVS

Indirekt:  $(nn)$

JMP

Abszolút indexelt (X):  $nn, X$

ADC/AND/ASL/CMP/DEC/EOR/INC/LDA/LDY/LSR/ORA/ROL/ROR/  
SBC/STA

Abszolút indexelt (Y):  $nn, Y$

ADC/AND/CMP/EOR/LDA/LDX/ORA/SBC/STA

Nullás lapu indexelt (X):  $\phi n, X$

ADC/AND/ASL/CMP/DEC/EOR/INC/LDA/LDY/LSR/ORA/ROL/ROR/  
SBC/STA/STY

Nullás lapu indexelt (Y):  $\phi n, Y$

LDX/STX

Indexelt indirekt:  $(\phi n, X)$

ADC/AND/CMP/EOR/LDA/DRA/SBC/STA

Indirekt indexelt:  $(\phi n), Y$

ADC/AND/CMP/EOR/LDA/DRA/SBC/STA

## 2. A jelzökre gyakorolt hatás szerint rendezve:

Eredménytől függően állítja:

N, Z

ADC/AND/ASL/BIT/CMP/CPX/CPY/DEX/DEY/EOR/INC/INX/INY/LDA/  
LDX/LDY/LSR/ORA/PLA/PLP/ROL/ROR/RTI/SBC/TAX/TAY/TYA/TSX/  
TXA

C

ADC/ASL/CMP/CPX/CPY/LSR/PLP/ROL/ROR/SBC

V

ADC/BIT/SBC

A jelzöt törli:

V

CLV

D

CLD

C

CLC

I

CLI

N

LSR

A jelzöt magasra állítja:

D

SED

C

SEC

I

BRK/SEI



## C. függelék

### KERNAL szubrutinok

- ACPTR - adatbyte input a soros buszról  
Belépési pont: \$FFA5  
Paraméterek: A - output/ az olvasott byte  
Előkészítő rutinok: TALK, TKSA  
Hívása:  
    JSR ACPTR  
    STA BYTE
- CHKIN - input csatorna előkészítése olvasásra  
Belépési pont: \$FFC6  
Paraméterek: X - input/ csatornaszám /filesorszám/  
Előkészítő rutinok: OPEN  
Hívása:  
    LDX #FILE  
    JSR CHKIN
- CHKOUT - output csatorna előkészítése olvasásra  
Belépési pont: \$FFC9  
Paraméterek: X - input/ csatornaszám /filesorszám/  
Előkészítő rutinok: OPEN  
Hívása:  
    LDX #FILE  
    JSR CHKOUT
- CHRIN - karakter olvasása az input csatornáról, ha nem előzte meg CHKIN, akkor a billentyűzetről a képernyőszerkesztő segítségével  
Belépési pont: \$FFCF  
Paraméterek: A - output/ beolvasott byte  
Előkészítő rutinok: OPEN, CHKIN  
Hívása:  
    JSR CHRIN  
    STA BYTE
- CHROUT - karakter küldése az output csatornára, ha nem előzte meg CHKIN, akkor a képernyő aktuális pozíciójára  
Belépési pont: \$FFD2  
Paraméterek: A - input/ kiírandó byte  
Előkészítő rutinok: OPEN, CHKOUT  
Hívása:  
    LDA BYTE  
    JSR CHRIN

CIOUT - +karakter küldése a soros buszra  
 Belépési pont: \$FFA8  
 Paraméterek: A - input/ kiírandó byte  
 Előkészítő rutinok: LISTEN, SECOND  
 Hívása:  
     LDA BYTE  
     JSR CIOUT

CINT - a képernyőszerkesztő és a VIC processzor inicializálása  
 Belépési pont: \$FFB1  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR CINT

CLALL - valamennyi csatorna és file lezárása  
 Belépési pont: \$FFE7  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR CLALL

CLOSE - file lezárása  
 Belépési pont: \$FFC3  
 Paraméterek: A - input/ filesorszám  
 Előkészítő rutinok: -  
 Hívása:  
     LDX #FILE  
     JSR CLOSE

CLRCHN - input és output csatorna lezárása és a csatornák eredeti (default) értékének visszaállítása  
 Belépési pont: \$FFCC  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR CLRCHN

GETIN - karakter olvasása a billentyűzet pufferből  
 Belépési pont: \$FFE4  
 Paraméterek: A - output/ beolvasott byte vagy 0, ha a puffer üres  
 Előkészítő rutinok: OPEN, CHKIN  
 Hívása:  
     CIKLUS JSR CHRIN  
     CMP #0  
     BEQ CIKLUS

IOBASE - input és output eszközök memóriafoglalása  
 Belépési pont: \$FFF3  
 Paraméterek: X - output/ a memóriacím alsó byte-ja  
               Y - output/ a memóriacím felső byte-ja

Előkészítő rutinok: -  
Hívása:  
JSR IOBASE

IOINIT - az input/output eszközök inicializálása  
Belépési pont: \$FFB4  
Paraméterek: -  
Előkészítő rutinok: -  
Hívása:  
JSR IOINIT

LISTEN - az input/output eszközt hallgató állapotba hozza  
Belépési pont: \$FFB1  
Paraméterek: A - input/ egységszám  
Előkészítő rutinok: -  
Hívása:  
LDA #EGYSEG  
JSR LISTEN

LOAD - program-file töltése/ellenőrzése a memóriába  
Belépési pont: \$FFD5  
Paraméterek: A - input/ 0=LOAD, 1=VERIFY  
ha az eszközt 0 másodlagos címmel  
nyitottuk meg  
X - input/ memóriacím alsó byte-ja  
Y - input/ memóriacím felső byte-ja  
Előkészítő rutinok: -  
Hívása:  
LDA #0  
JSR LOAD

MEMBOT - beállítja a BASIC ROM kezdőcímét  
Belépési pont: \$FF9C  
Paraméterek: X - input/ memóriacím alsó byte-ja  
Y - input/ memóriacím felső byte-ja  
Az alapfeltételezés szerinti kezdőérték \$800  
Előkészítő rutinok: -  
Hívása:  
JSR MEMBOT

MEMTOP - beállítja a BASIC ROM végcímét  
Belépési pont: \$FF99  
Paraméterek: X - input/ memóriacím alsó byte-ja  
Y - input/ memóriacím felső byte-ja  
Előkészítő rutinok: -  
Hívása:  
JSR MEMTOP

OPEN - megnyit egy file-t  
Belépési pont: \$FFC0  
Paraméterek: -  
Előkészítő rutinok: SETLFS, SETNAM  
Hívása:  
JSR OPEN

PLOT *C=0-* beállítja a cursor pozícióját *C=1 lekérdezi*  
 Belépési pont: \$FFFO  
 Paraméterek: X - input/ sor (0-24)  
                   Y - input oszlop (0-39)  
 Előkészítő rutinok: SETLFS, SETNAM  
 Hívásai:  
           LDX #SOR  
           LDY #OSZLOP  
           JSR PLOT

RAMTES - memóriateszt, majd a memóriában elhelyezett rendszerpointerek és munkaterületek inicializálása  
 Belépési pont: \$FFB7  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívásai:  
           JSR RAMTES

RDTIM - kiolvassa a rendszer-órát  
 Belépési pont: \$FFDE  
 Paraméterek: Y - output/ óra alsó byte-ja  
                   X - output/ óra középső byte-ja  
                   A - output/ óra felső byte-ja  
 Előkészítő rutinok: -  
 Hívásai:  
           JSR RDTIM  
           STY ORA  
           STX ORA+1  
           STA ORA+2

READST - kiolvassa az input/output állapotszót  
 Belépési pont: \$FFB7  
 Paraméterek: A - output/ állapotszó  
 Előkészítő rutinok: -  
 Hívásai:  
           JSR READST

RESTOR - visszaállítja eredeti értékeire a rendszervektorokat  
 Belépési pont: \$FFBA  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívásai:  
           JSR RESTOR

SAVE - program-file kimentése a memóriából  
 Belépési pont: \$FFD8  
 Paraméterek: A - input/ egy nullás lapon lévő cím, amely a program kezdetére mutat  
                   X - input/ a végcím alsó byte-ja  
                   Y - input/ a végcím felső byte-ja  
 Előkészítő rutinok: SETLFS, SETNAM

Hívása:

```
LDA ELEJE
LDX #VEGE<
LDY #VEGE>
JSR SAVE
```

SCNKEY - az éppen megnyomott billentyűnek megfelelő ASCII kódot a billentyűzet pufferébe továbbítja

Belépési pont: \$FF9F

Paraméterek: -

Előkészítő rutinok: IOINIT

Hívása:

```
JSR SCNKEY
```

SCREEN - a képernyő méretét adja vissza a nagyobb Commodore gépeken is futó programokkal való kompatibilitás miatt

Belépési pont: \$FFED

Paraméterek: X - output/oszlopok száma

Y - output/sorok száma

Előkészítő rutinok: -

Hívása:

```
JSR SCREEN
STX MAXO
RTY MAXS
```

SECOND - másodlagos cím küldése a hallgató állapotú eszköz felé

Belépési pont: \$FF93

Paraméterek: A - input/ másodlagos cím

Előkészítő rutinok: LISTEN

Hívása:

```
LDA #MASODL
JSR SECOND
```

SETLFS - file-paraméterek beállítása

Belépési pont: \$FFBA

Paraméterek: X - input/ filesorszám

Y - input/ másodlagos cím, ha nincs \$FF

A - input/ egységszám

Előkészítő rutinok: -

Hívása:

```
LDX #FILE
LDY #$FF
LDA #DEV
JSR SETLFS
```

SETMSG - üzenetek kiírását engedélyezi illetve tiltja le. Az akkumulátor legmagasabb helyiértékű bitje a hibaüzenetekre (pl. FILE NOT FOUND), a következő bit a vezérlő üzenetekre (pl. PRESS PLAY ON CASSETTE) vonatkozik. Ritkán használatos.

Belépési pont: \$FF90

Paraméterek: A - input / 7. bit - hibaüzenet  
6. bit - vezérlő üzenet

Előkészítő rutinok: -

Hívása:

```
LDA #$80 ; hibaüzenetek engedélyezése
JSR SETMSG
```

144

TKSA - másodlagos cím küldése a beszélő állapotú eszköz felé  
 Belépési pont: \$FF96  
 Paraméterek: A - input/ másodlagos cím  
 Előkészítő rutinok: TALK  
 Hívása:  
     LDA #MASODL  
     JSR TKSA

UDTIM - aktualizálja - egy egységgel növeli - a rendszer-órát, csak akkor kell hívni, ha a megszakításokat magunk kezeljük  
 Belépési pont: \$FFEA  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR UDTIM

UNLSN - valamennyi hallgató állapotú input/output eszközön leállítja  
 Belépési pont: \$FFAE  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR UNLSN

UNTLK - valamennyi beszélő állapotú input/output eszközön leállítja  
 Belépési pont: \$FFAB  
 Paraméterek: -  
 Előkészítő rutinok: -  
 Hívása:  
     JSR UNTLK

VECTOR - a memóriában lévő rendszer-vektorok kezdőcímének lekérdezése/beállítása  
 Belépési pont: \$FF8D  
 Paraméterek: C - input/ magas -> lekérdezés  
                   X - output/ a cím alsó byte-ja  
                   Y - output/ a cím felső byte-ja  
           vagy C - input/ alacsony -> beállítás  
                   X - input/ a cím alsó byte-ja  
                   Y - input/ a cím felső byte-ja  
 Előkészítő rutinok: -  
 Hívása:  
     LDX #VEC<  
     LDY #VEC>  
     CLC  
     JSR VECTOR  
     vagy  
     SEC  
     JSR VECTOR  
     STX VECCIM  
     STY VECCIM+1

Ajánlott és részben felhasznált irodalom:

1. Ury L.: Commodore 64 BASIC felhasználó kézikönyv  
(LSI ATSZ, BP. 1984)
2. Commodore 64 Programmer's Reference Guide  
Commodore Business Machines, Inc. 1982 angol nyelven
3. Angerhausen, M.; Becker, A. et al.: The Anatomy of the  
Commodore 64  
Abacus Software, Grand Rapids 1984 angol nyelven  
vagy  
Angerhausen, M.; Becker, A. et al.: Commodore 64 Intern  
Data Becker, Düsseldorf 1983  
(fentivel megegyező kiadás) német nyelven
4. French, D.: Inside the Commodore 64  
French Silk, Minneapolis 1984 angol nyelven
5. Greeshields, M.: 40 Best Machine Code Routines For  
Commodore 64  
Duckworth, London, 1984 angol nyelven
6. Stephenson, A.P. and Stephenson, D.J.: Advanced Machine  
Code Programming for the Commodore 64  
Granada, London 1984 angol nyelven
7. Butterfield, J.: Machine Language for the Commodore 64  
and Other Commodore Computers  
Brady Communications Company, Inc. Bowie 1984  
angol nyelven
8. English, L.: Das Maschinensprache Buch zum Commodore 64  
Data Becker, Düsseldorf 1984 német nyelven
9. Lawrence, D. and England, M.: Commodore 64 Machine Code  
Master  
Sunshine Books, London 1983 angol nyelven



**Készült: MÁTRAINVEST GT Nyomda**  
**Felelős kiadó: Kovács Magda**  
**Felelős vezető: dr. Csák Máté**  
**Terjedelem: 18,5 B/5 ív**  
**Engedély szám: 53629**

## **COMMODORE-64 oktatócsomag**

Az LSI ATSZ legújabb kínálata a **Commodore 64** oktatócsomag, ami mind a Commodore 64 személyi számítógép napi használatához, mind programozásának elsajátításához szükséges ismereteket tartalmazza.

**Az oktatócsomag az alábbiakat tartalmazza:**

**dr. Üry László: Commodore-64 BASIC felhasználói kézikönyv**

A kézikönyv második bővített, kiadása a C-64 BASIC-jén túlmenően részletesen ismerteti a SIMONS' BASIC, a SUPERGRAPHIC utasításkészletét, a BASIC 4.0-ás bővítést. Tartalmazza a C-64 grafikájának, illetve hanggenerátorának használatához szükséges ismereteket. A könyv ismerteti a C-64-hez hazánkban vásárolható valamennyi típusú lemezegység használatát, beleértve az IEEE-illesztésű lemezegységeket is.

**Erdős Iván: A Commodore-64 assembly nyelvű programozása**

A könyv részletesen ismerteti az M6510-es utasításkészleteét. Részletesen tárgyalja a Commodore-64 assembly programozását segítő szoftver eszközöket, s bőséges példanyaggal segíti a témával most ismerkedőket. Ismerteti a C-64 ROM-jának legfontosabb részeit.

**dr. Üry László: Commodore-64 Információs kártya**

Az információs kártya a Commodore-64 használatához szükséges ismereteket foglalja össze táblázatos formában, beleértve a BASIC szintaxisát, a BASIC és a DOS hibaüzeneteket és az M6510-es processzor teljes utasításkészletét.

### **Commodore oktatólemez**

Az oktatólemez az előző két könyv példanyagán túl további kidolgozott assembly és BASIC programozási példákat tartalmaz.

**Az oktatócsomag ára: 3500 Ft.**



Speciális igényével forduljon speciális szaküzlethez;

## APISZ SZÁMÍTÁSTECHNIKAI SZAKÜZLETEI



Budapest VIII., Szigony u. 15.

Telefon: 143—446

Telex: 22—7803

Budapest XI., Budafoki út 7.

Telefon: 665—503

(APISZ és SZÁMALK közös boltja)

### Raktárról kapható:

- |                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>— mágnescsíkos kartonok A/4 fekvő, álló, kvadrát kivitelben,</li><li>— leporellók, közép- és számítógépekhez,</li><li>— pénzügyi leporellók,</li><li>— pelikán kazettás írógépszalag,</li><li>— carbonszalag, festéklepedő,</li><li>— kézi adatfeldolgozáshoz készülékek és kártyák,</li><li>— mágneses diszpozíciós táblák készletekkel,</li><li>— kartontároló kocsi és szekrény,</li></ul> | <ul style="list-style-type: none"><li>— számítástechnikai médiák, tartozékok,</li><li>— mágneses háttértárak (mágnesszalagok, lemezek, floppyk),</li><li>— festékszalagok, festékkendők,</li><li>— számítástechnikai könyvek,</li><li>— speciális íróeszközök: fóliára író filcironok, vonalzők, sablonok,</li><li>— software termékek,</li><li>— számítástechnikai szolgáltatások,</li></ul> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

és még sok más

**SZAKMAI TANÁCSADÁS**  
**MIND A MAGÁNVÁSÁRLÓK, MIND A VÁLLALATOK, INTÉZMÉNYEK**  
**RENDELKEZÉSÉRE ÁLLUNK**